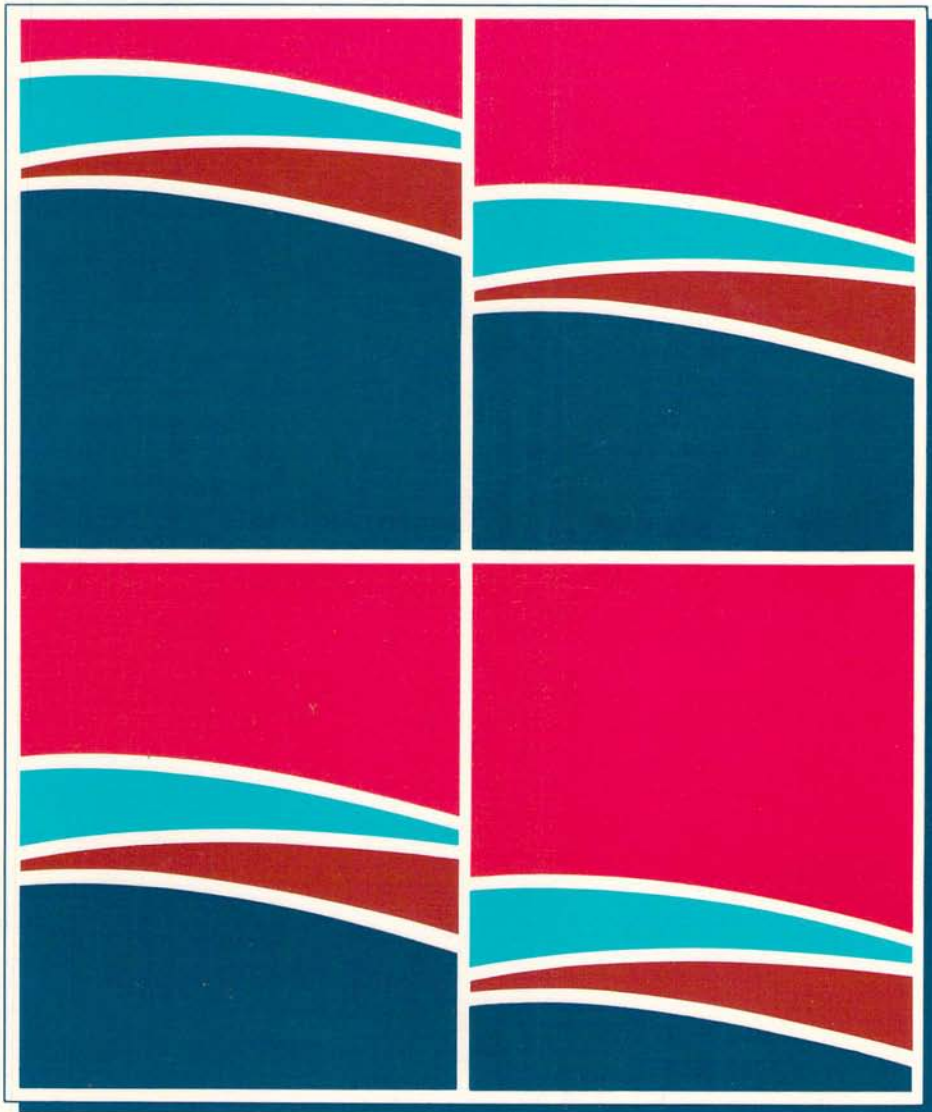


# ***TURBO Pascal 4.0/5.0***

*An Introduction to the Art and Science of Programming*



*Walter J. Savitch*

## TURBO Pascal Program Layout

```
{ $R+ compiler directive for array index checking. }
program Sample;
{ TURBO Pascal sample program layout. }
uses crt, MyStuff; { Omit if no units are used. }
const Low = 0;
      High = 100;
type Word = string[10];
      Index = Low .. High;
      List = array[Index] of integer;
var N, M: integer;
    Result: real;
    Count: integer;
    Answer: Word;
    A, B: List;

procedure Doubler(X: integer; var Y: integer);
{ Sets Y equal to twice the value of X. }
begin {Doubler}
    Y := 2*X
end; {Doubler}

function Average(N1, N2: integer): real;
var Sum: integer; {Local variable}
begin {Average}
    Sum := N1 + N2;
    Average := Sum/2 {Value returned is Sum/2.}
end; {Average}

procedure Sample(X, Y: real; var Name: Word; var A: List);
var Temp: real; {Local variable}
begin {Sample}
    . . .
end; {Sample}

begin {Program}
    . . .
    {Procedure call that sets N equal to twice the value of (M + 7): }
    Doubler (M + 7, N);
    . . .
    {Function call that sets Result equal to the Average of (M + 2) and N: }
    Result := Average (M + 2, N);
    . . .
end. {Program}
```



# ***TURBO Pascal 4.0/5.0***

***An Introduction to  
the Art and Science  
of Programming***

G. Booch

**Software Components with Ada: Structures, Tools, and Subsystems**

G. Booch

**Software Engineering with Ada, Second Edition**

G. Brookshear

**Computer Science: An Overview, Second Edition**

G. Brookshear

**Theory of Computation: Formal Languages, Automata and Complexity**

D. M. Etter

**Structured FORTRAN 77 for Engineers and Scientists, Second Edition**

D. M. Etter

**Problem Solving with Structured FORTRAN 77**

D. M. Etter

**Problem Solving in Pascal for Engineers and Scientists**

C. Fischer and R. Leblanc

**Crafting a Compiler**

P. Helman and R. Veroff

**Intermediate Problem Solving and Data Structures**

P. Helman and R. Veroff

**Walls and Mirrors: Modula-2 Edition**

N. Miller

**File Structures Using Pascal**

A. Kelley and I. Pohl

**A Book on C**

A. Kelley and I. Pohl

**C by Dissection**

A. Kelley and I. Pohl

**TURBO C: Essentials of C programming**

W. J. Savitch

**Pascal: An Introduction to the Art and Science of Programming, Second Edition**

W. J. Savitch

**Turbo Pascal 4.0/5.0**

R. Sebesta

**Concepts of Programming Languages**

M. Sobell

**A Practical Guide to the UNIX System, Second Edition**

# ***TURBO Pascal 4.0/5.0***

***An Introduction to  
the Art and Science  
of Programming***

***Walter J. Savitch***

*University of California, San Diego*



**The Benjamin/Cummings Publishing Company, Inc.**

Redwood City, California • Fort Collins, Colorado

Menlo Park, California • Reading, Massachusetts • New York

Don Mills, Ontario • Wokingham, U.K. • Amsterdam • Bonn

Sydney • Singapore • Tokyo • Madrid • San Juan

Sponsoring Editor: Alan R. Apt, Associate Editor: Mark McCormick  
Production Editors: Mary Shields, Eleanor Renner Brown  
Interior Design Modifications: Seventeenth Street Studios  
Cover Design: Seventeenth Street Studios  
Illustrations: Sally Shimizu  
Composition: G & S Typesetters, Inc.

Acknowledgments:

Chapter 2, opening quotation from E. W. Dijkstra, *Notes on Structured Programming*. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Academic Press (1972). Reprinted by permission of the author and publisher.

Chapter 7, ending quotation from F. P. Brooks, Jr., *The Mythical Man-Month, Essays on Software Engineering*, p. 116. Addison-Wesley Publishing Company (1975). Reprinted by permission of the publisher.

Chapter 11, ending quotation from Niklaus Wirth, *Algorithms + Data Structures = Programs*, title. Prentice-Hall (1976). Reprinted by permission of the publisher.

Chapter 14, opening quotation from John R. Ross, *Constraints on Variables in Syntax*, p.i., Ph.D. dissertation, Massachusetts Institute of Technology (1967). Reprinted by permission of the author.

Chapter 14, midchapter quotation from Jorge Luis Borges, "The Garden of Forking Paths," in Jorge Luis Borges, *Selected Stories and Other Writings*, p. 25. New Directions Publishing Company (1964). Reprinted by permission of the publisher.

Chapter 15, ending quotation from B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, 2nd ed., p. 117. McGraw-Hill Book Co. (1978). Reprinted by permission of the publisher and Bell Laboratories.

IBM, XT, AT, and PS/2 are trademarks of International Business Machines, Inc.; MS-DOS is a trademark of Microsoft, Inc.; TURBO Pascal is a trademark of Borland International, Inc.; WordStar is a trademark of MicroPro International, Inc.

Copyright © 1989 by The Benjamin/Cummings Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

The programs presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs.

Library of Congress Cataloging-in-Publication Data

Savitch, Walter J., 1943—

Turbo Pascal 4.0/5.0 : an introduction to the art and science of programming / Walter J. Savitch.

p. cm.—(The Benjamin/Cummings series in structured programming)

Includes index.

ISBN 0-8053-0410-X

1. PASCAL (Computer program language) 2. Turbo Pascal (Computer program) I. Title. II. Series.

QA76.73.P2S285 1989

005.13'3—dc19

88-30598

CIP

DEFGHIJ-HA-89321098

The Benjamin/Cummings Publishing Company, Inc.  
390 Bridge Parkway  
Redwood City, CA 94065



---

*To ir526*

---





---

# *Preface*

This book was designed for use in introductory computer science and programming classes that use the TURBO Pascal language. It can be used for courses as short as one quarter or as long as one academic year. The book includes a thorough introduction to problem solving and programming techniques as well as complete descriptions of both versions 4.0 and 5.0 of the TURBO Pascal language. It assumes no knowledge of computers and no mathematics beyond high school algebra.

This is the fifth in a series of books I have written featuring the Pascal language. Each of the books is an introduction to computer science and programming. Each new edition was written to accommodate newly developed techniques for designing programs and newly introduced features in the Pascal language. Version 4.0 of TURBO Pascal represents a significant expansion of the Pascal language providing new facilities for handling procedural and data abstraction. Version 5.0 adds a source code level interactive debugger. This latest text was written to include complete coverage of these new features and the techniques for using them effectively. Additional details about the book are summarized below.

---

## **Thorough Coverage of Problem Solving Techniques**

The text includes complete coverage of problem solving techniques and illustrates these techniques with numerous case studies showing the complete design process in detail from problem definitions to final Pascal program.

---

## **Thorough Coverage of Debugging and Programming Techniques**

Thorough coverage of debugging and programming techniques is presented throughout the text. Boxed sections on common *Pitfalls* highlight important techniques in a compact easy to find and easy to digest way. An introduction to the version 5.0 interactive debugger is included in an appendix. For those students who are using version 5.0, this introduction makes the debugger accessible as soon as they have written their first few programs. Other sections of the appendix explain how to use the debugger with more advanced Pascal constructions.

---

---

## Early Introduction of Procedures

As with previous texts in this series, this book introduces procedures very early and presents a complete discussion of parameters as soon as procedures are introduced. Both value and variable parameters are introduced immediately after introducing the notion of a procedure.

It had been my experience in teaching introductory programming classes that students find parameters easier to understand when the topic is presented early. Moreover they develop a better programming style as a result. Users of previous texts in this series have confirmed the results of this experiment. What was experimental in the first of these texts is now widely accepted as sound pedagogy.

---

## Extensive Array Coverage

Two full chapters are dedicated to arrays. These chapters cover both one-dimensional and multidimensional arrays, with heavy emphasis on programming techniques such as reading data into arrays, manipulating partially filled arrays, and designing data structures using arrays. The array coverage is followed by a chapter with complete coverage of records. This permits covering records immediately after arrays (or later, if that is preferred).

---

## Extensive Coverage of Recursion

An entire chapter is dedicated to recursion. Extensive use of figures and examples makes this difficult topic accessible to beginners.

---

## Self-Test and Summary Sections

*Self-Test Exercises*, with answers in the back of the book, are provided throughout the text. Each chapter contains a complete *Summary of Problem Solving and Programming Techniques* as well as a complete *Summary of Pascal Constructs*. The Pascal summaries include templates and typical examples for quick and easy reference. These summaries cover both TURBO Pascal and standard Pascal.

---

## Complete Coverage of Versions 4.0 and 5.0 of TURBO Pascal

Versions 4.0 and 5.0 of TURBO Pascal are essentially the same, except that version 5.0 has more features. Versions 4.0 and 5.0 both differ from the previous 3.0 version primarily in two ways: They have a new menu environment and a new TURBO Pascal fea-

---

ture called *units*. This book provides complete coverage of TURBO Pascal, including these new features, as an integral part of the text. The main feature that was added to version 5.0 is an interactive debugger. This debugger is covered in an appendix that is designed so that it can be read along with the main body of the text.

Units are a new feature of TURBO Pascal available in versions 4.0 and 5.0. They allow for separate compilation of modules. This allows programmers to build up libraries of their own predefined procedures and functions. Units also facilitate faster compilation of programs and allow for larger program sizes than version 3.0. However, the main advantages of units are that they allow the building of procedure libraries and that they facilitate procedural and data abstraction. It is these advantages which are emphasized in this text. The text includes thorough coverage of TURBO Pascal units including examples and case studies to illustrate procedural and data abstraction.

As every TURBO Pascal instructor knows, learning to write TURBO Pascal programs involves more than the Pascal language and programming technique. Before learning to program in TURBO Pascal one must become familiar with the TURBO editor and other TURBO utilities, and in most cases, with the DOS operating system. A student typically needs a DOS manual and a TURBO environment manual in addition to a TURBO Pascal text. Both of these are included as Appendixes 5 through 10 of this book. They include introductions to the DOS operating system, the TURBO editor, and the TURBO menu system. They can be read, in order, as a coherent and complete guide for the beginning student. Emphasis is placed on the DOS operating system as used on the IBM PC, PS/2, and similar machines. However, the appendixes are sufficiently self-contained and versatile to accommodate other users.

Version 5.0 of TURBO Pascal includes a powerful interactive debugger. Appendix 18 includes a tutorial introduction to the debugger which is accessible to students as soon as they begin to study the Pascal language. Later sections of that appendix explain how to use the debugger with more advanced constructions such as arrays and units.

Aside from the interactive debugger, which has been added to the version 5.0 environment, there are no substantial differences between versions 4.0 and 5.0 of TURBO Pascal. In those rare instances when the two versions do differ in some details, the text includes separate sections for the two versions. All of the programs in the text have been compiled and run both using version 4.0 and using version 5.0.

Brief coverage of standard Pascal features that differ from TURBO Pascal is included. However, the standard Pascal sections are optional; when the two dialects differ, the TURBO Pascal features are used in the case studies. The result is a true TURBO Pascal text, but with just enough standard Pascal coverage to teach portable programming.

---

## **Allows Early or Late Introduction of Units**

Units are not as critical to modular design as procedures are. One can design truly modular programs without using the new TURBO Pascal feature of units. However, units do provide a way of emphasizing, and even enforcing, the technique of modular design, as well as providing other important design and pragmatic advantages. Hence, it makes sense to cover units fairly early. The chapter on units is divided into two parts so

---



as to allow flexibility and still allow units to be introduced as early (or as late) as an instructor might desire. The first part of the chapter covers all the material a student needs in order to write units. This part may be covered anytime after function declarations are introduced. The second part covers no new TURBO Pascal, but concentrates on data abstraction techniques. This second part may be covered at any time after the introduction of arrays.

---

## Advanced Topics

This text includes material suitable for a course such as the ACM CS1 course or the Educational Testing Service's Advanced Placement course. It also includes coverage of a number of more advanced computer science topics that are not normally covered in a first course. These include procedural abstraction, data abstraction, sorting and searching, hashing, numeric programming, loop invariants, dynamic data structures, and trees. By choosing some or all of these topics an instructor can create a course of almost any size or level.

---

## Flexibility

The order in which topics can be covered is extremely flexible. The material on the TURBO environment has been broken down into a series of appendixes which can be read at the beginning of the course as a single unit or can be interspersed with topics in Pascal and problem solving. Units may be introduced either early or late in a course.

The chapter on text files is divided into two parts to allow for two possibilities, either postponing the topic entirely until later in a course or briefly introducing them early and giving more detail later on.

Most advanced topics—such as recursion, some software engineering topics, some numeric programming techniques, and a substantial amount on data structures including records, files, and pointers—are packaged into chapters that can be covered in almost any order. Alternatively, a subset of the chapters may be chosen to form a shorter course. To add even more flexibility sections with optional topics are included throughout the book. The dependency chart at the front of this book shows the possible orders in which the chapters can be covered without losing continuity.

---

## Support Material

A chapter-by-chapter instructor's guide is available from the publisher as are diskettes with tutorial software and diskettes containing all the programs from the text.

---

## Acknowledgments

I have received much help and encouragement from numerous individuals and groups while preparing this series of Pascal books. Much of the original edition was written

---



while I was visiting at the University of Washington (Seattle) Computer Science Department. Parts of the earlier TURBO Pascal texts were written while I was visiting the Computer Science Department at the University of Cincinnati. The remainder of the work on these books was done in the Computer Science and Engineering Department at the University of California, San Diego (UCSD). I am grateful to these three institutions for providing facilities and a conducive environment for writing these books.

I extend my thanks to Lieutenant Commander Paul Desilets for helpful discussions on the new features of version 4.0 of TURBO Pascal. Major Scott C. Teel was kind enough to review an earlier version of the material on these new features. His comments contributed much to the final version and I am very grateful for his help.

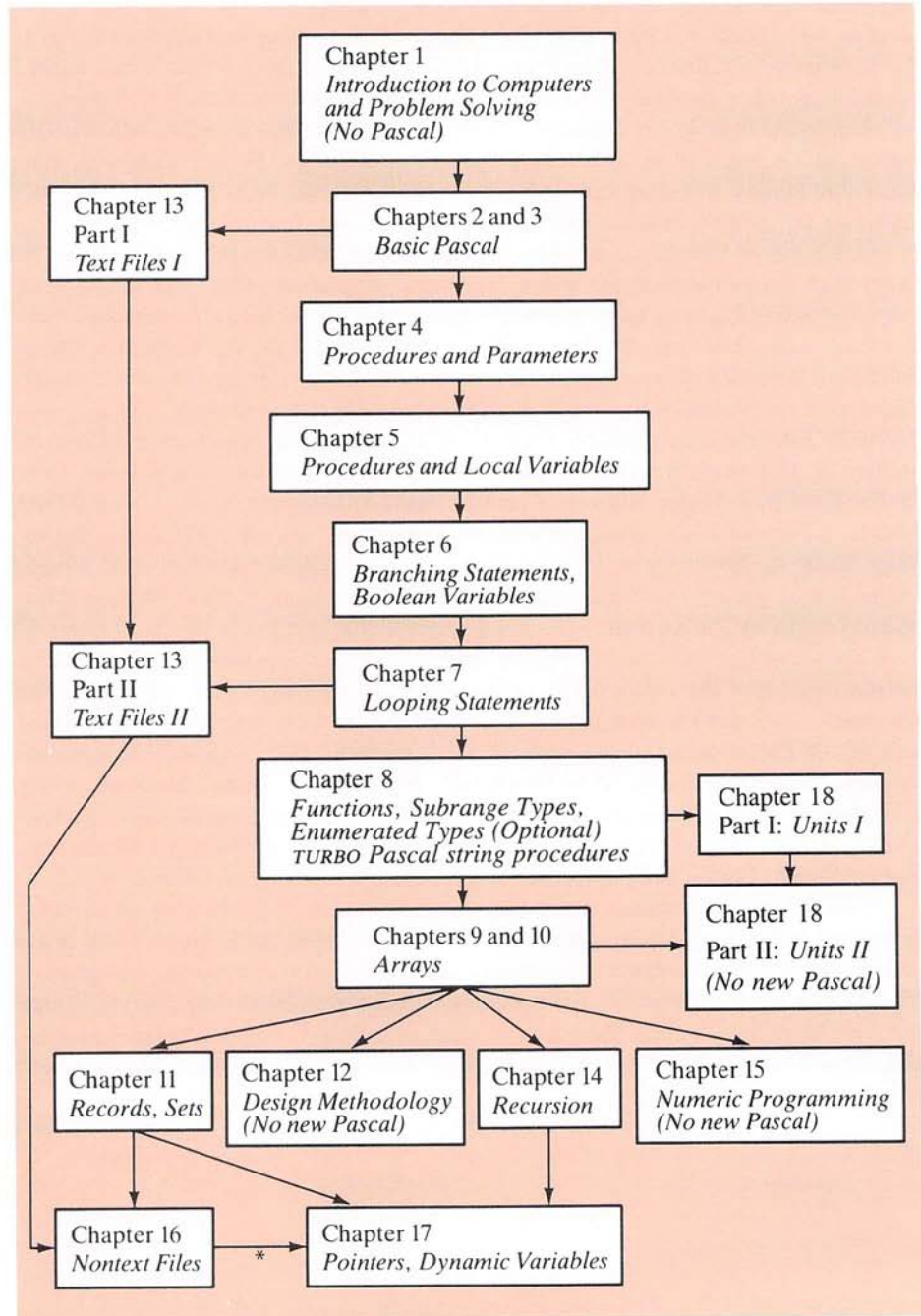
A number of individuals provided reviews or comments for this and/or the previous TURBO Pascal books in this series. Their suggestions and comments were a great help in shaping this text, and I gratefully acknowledge their help. In alphabetical order these individuals are: Stephen Andrilli, Owen Astrachan, R. W. Barton, Philip Beckman, Beverly Bilshausen, Baldwin van der Byl, Phillip Carrigan, Scott Cormode, Christine Coulter, Molly E. Daniel, H. Edward Donley, Chris Dovolis, Eileen Entin, David R. Falconer, Allyn Fratkin, Capt. C. C. Gardner, Chia Yung Han, Paul Hanna, Robert M. Holloway, Bill Hotard, Dale W. Isner, Brian Johnson, Ken Kellum, Robert M. Knodel, J. Mailen Kootsey, Karl Krummel, N. Lehmkuhl, K. W. Loach, Moira Mahony, Michael Main, Rayno Niemi, G. Ozsoyogui, Jerome Paul, James Payne, Gary Phillips, Colette Pirie, Howard Pyron, Major W. A. Richardson, Ned Rosen, Arden Ruttan, George Stockman, Rena Tobias, Joseph Waters, Gregory Wetzel, Anne Wilson, and Guy Zimmerman.

The following individuals provided varying forms of suggestions and comments on one or more of the earlier books in this series and their marks are still apparent in this book. Their contributions ranged from technical discussions to reviewing drafts of one of the earlier books. I am grateful to all of them: Guy Almes, Bill Appelbe, Andrew Black, Jim Bunch, Mike Denisevich, John Donald, Patrick Dymond, Klaus Eldridge, Jim Gips, Richard Kaufmann, Alean Kirnak, Gregg Kornfeld, Keith Muller, Dorian Naveh, Vijay Rao, J. F. Paris, Robert Rother, Gary Sackett, Joe Sandmeyer, Robert Streett, Martin Tompa, Dennis Volper, Chin Wu, and Larry Weber.

I extend a special thanks to the many students in my programming classes who tested and helped correct preliminary versions of all editions in this series. I also thank all the individuals at Benjamin/Cummings who organized the reviewing and production of this book. In particular, Eleanor Renner Brown, Mark McCormick, Laura Kenney, Mary Shields, and Mary Ann Telatnik contributed much to the finished product. Finally, I thank my editor Alan Apt for his excellent help and support throughout this entire series of books.

W. J. S.

## Dependency Chart of Chapters and Location of Key Pascal Constructs



\*Two case studies in Chapter 17 use files.  
The remainder of Chapter 17 does not use files.

---

## ***Brief Contents***

Chapter One	<b>Introduction to Computers and Problem Solving</b>	1
Chapter Two	<b>Introduction to Problem Solving with Pascal</b>	21
Chapter Three	<b>More Pascal and Programming Techniques</b>	65
Chapter Four	<b>Designing Procedures for Subtasks</b>	105
Chapter Five	<b>Procedures for Modular Design</b>	143
Chapter Six	<b>Designing Programs That Make Choices</b>	179
Chapter Seven	<b>Problem Solving Using Loops</b>	217
Chapter Eight	<b>Designing Functions and Data Types</b>	267
Chapter Nine	<b>Arrays for Problem Solving</b>	331
Chapter Ten	<b>Complex Array Structures</b>	373
Chapter Eleven	<b>Records and Other Data Structures</b>	425
Chapter Twelve	<b>Program Design Methodology</b>	479
Chapter Thirteen	<b>Text Files and Secondary Storage</b>	503
Chapter Fourteen	<b>Problem Solving Using Recursion</b>	541
Chapter Fifteen	<b>Solving Numeric Problems</b>	579
Chapter Sixteen	<b>More File Types</b>	605
Chapter Seventeen	<b>Dynamic Data Structures</b>	635
Chapter Eighteen	<b>Units for Modularization and Abstraction</b>	691
Appendix One	<b>The goto and exit Statements</b>	A.1
Appendix Two	<b>TURBO Pascal—More Integer Types</b>	A.5
Appendix Three	<b>Precedence of Operators</b>	A.6
Appendix Four	<b>Syntax Diagrams for TURBO Pascal Programs</b>	A.7
Appendix Five	<b>A Brief Introduction to the IBM PC and PC-DOS</b>	A.22
Appendix Six	<b>Summary of DOS Commands</b>	A.34
Appendix Seven	<b>A Quick Introduction to the TURBO Menu Environment, with Emphasis on the Editor</b>	A.36
Appendix Eight	<b>The TURBO File Menu</b>	A.49
Appendix Nine	<b>TURBO Hotkeys</b>	A.53

---

Appendix Ten	<b>The TURBO Editor Commands</b>	A.55
Appendix Eleven	<b>Compiling to Disk in TURBO</b>	A.58
Appendix Twelve	<b>Input/Output Redirection in DOS</b>	A.60
Appendix Thirteen	<b>The TURBO Crt Unit</b>	A.62
Appendix Fourteen	<b>TURBO Pascal—Character Graphics</b>	A.66
Appendix Fifteen	<b>TURBO Pascal—Text Windows</b>	A.69
Appendix Sixteen	<b>Some Common TURBO Pascal Compiler Directives</b>	A.72
Appendix Seventeen	<b>TURBO Pascal—I/O Error Handling</b>	A.75
Appendix Eighteen	<b>Version 5.0—The Integrated Debugger</b>	A.77
Appendix Nineteen	<b>ASCII Character Set</b>	A.86
Glossary		A.88
Answers to Self-Test Exercises		A.97
Index		I.1
Inside Back Cover		
Reserved Words		
Ordinal Types		
Types that May Be Used for the Value Returned by a Function		
Ranges for TURBO Pascal Numbers		

---



---

# Contents

## CHAPTER ONE

### Introduction to Computers and Problem Solving 1

---

What a Computer Is	3
The Modern Digital Computer	3
The Notion of an Algorithm	8
Programs and Data	9
High Level Languages	10
The Pascal Language	12
Programming Environments	13
Designing Programs	14
Summary of Terms	17
Exercises	18
References for Further Reading	20

## CHAPTER TWO

### Introduction to Problem Solving with Pascal 21

---

The Notion of a Program Variable	23
Stepping Through a Program	23
Assignment Statements	26
<i>Pitfall—Uninitialized Variables</i>	27
Data Types—An Introduction	28
More about real Values	31
Type Compatibility	32
Arithmetic Expressions	32
Simple Output	35
Input	36
Designing Input and Output	38
<i>Pitfall—Input in Wrong Order</i>	38
Names: Identifiers	38
Putting the Pieces Together	40
<i>TURBO Pascal</i>	41
Introduction to Programming Style	46

---



Blaise Pascal (Optional)	47
Self-Test Exercises	48
Interactive Exercises	49
Problem Solving and Program Design	49
Top-Down Design	50
<i>Case Study—A Guessing Game</i>	50
Integer Division— <i>mod</i> and <i>div</i>	51
Desktop Testing	53
<i>Case Study—Making Change</i>	53
Exploring the Solution Space	58
Summary of Problem Solving Techniques	58
Summary of Programming Techniques	59
Summary of Pascal Constructs	59
Exercises	62

### CHAPTER THREE

## More Pascal and Programming Techniques

65

Naming Constants	67
Comments	68
<i>Pitfall—Forgetting a Closing Comment Delimiter</i>	69
Formatted Output	69
Example Using Named Constants and Formatted Outputs	71
Allowable Range for Numbers	72
More about Commenting	73
Testing and Debugging	74
Tracing	77
Use of Assertions in Testing (Optional)	78
Self-Test Exercises	79
Interactive Exercises	79
Syntax Diagrams	79
Simple Branching— <i>if-then-else</i>	81
<i>Pitfall—The Equality Operator</i>	84
Optional <i>else</i>	85
<i>Pitfall—Extra Semicolons</i>	87
Compound Statements	87
Iterative Enhancement	88
<i>Case Study—Payroll Calculation</i>	88
Standard Functions	92
Using Known Algorithms	93
<i>Case Study—Solving Quadratic Equations</i>	93
Defensive Programming	96
More about Indenting and Commenting	96
Summary of Problem Solving Techniques	96
Summary of Programming Techniques	97
Summary of Pascal Constructs	97

Exercises	100
References for Further Reading	103

## CHAPTER FOUR

## Designing Procedures for Subtasks 105

Simple Pascal Procedures	107
Variable Parameters	107
Parameter Lists	109
Implementation of Variable Parameters (Optional)	112
Procedures Calling Procedures	114
Procedural Abstraction	118
Self-Test Exercises	119
Value Parameters	121
What Kind of Parameter to Use	123
Mixed Parameter Lists	124
<i>Case Study—Supermarket Pricing</i>	126
<i>Case Study—Change Program with Procedures</i>	128
<i>Pitfall—Incorrectly Ordered Parameter Lists</i>	132
Generalizing Procedures	133
Choosing Parameter Names	134
Summary of Problem Solving and Programming Techniques	136
Summary of Pascal Constructs	136
Exercises	138

## CHAPTER FIVE

## Procedures for Modular Design 143

Local Variables	145
<i>Case Study—Grade Warnings</i>	147
Other Local Identifiers	151
<i>Pitfall—Use of Global Variables</i>	152
Self-Test Exercises	154
Implementation of Value Parameters	155
<i>Pitfall—Inadvertent Local Variables</i>	157
Scope of an Identifier	157
<i>Case Study—Automobile Bargaining</i>	162
Testing Procedures	165
Top-Down and Bottom-Up Strategies	165
Preconditions and Postconditions	171
<i>Case Study—Calculating Leap Years</i>	172
Summary of Problem Solving and Programming Techniques	174
Summary of Pascal Constructs	174
Exercises	175

## CHAPTER SIX

**Designing Programs that Make Choices****179**

Nested Statements	181
Nesting <i>if-then</i> and <i>if-then-else</i> Statements	182
Complex Boolean Expressions	182
George Boole (Optional)	185
Evaluating Boolean Expressions	185
<i>Pitfall—Undefined Boolean Expressions</i>	186
Self-Test Exercises	187
Programming with Boolean Variables	188
<i>Pitfall—Omitting Parentheses in Boolean Expressions</i>	189
Boolean Input and Output	191
<i>Case Study—Designing Output</i>	191
Boolean Constants and Debugging Switches	193
The <i>case</i> Statement	196
<b>TURBO Pascal—The Case Statement</b>	197
The Empty Statement	201
Programming Multiple Alternatives	202
<i>Case Study—State Income Tax</i>	203
<i>Case Study—Scoring Blackjack</i>	204
Summary of Problem Solving and Programming Techniques	210
Summary of Pascal Constructs	211
Exercises	213

## CHAPTER SEVEN

**Problem Solving Using Loops****217**

A Case Study Introducing Loops	219
The <i>while</i> Statement	220
<i>Pitfall—Uninitialized Variables</i>	222
<i>Pitfall—Unintended Infinite Loops</i>	223
Self-Test Exercises	224
Terminating an Input Loop	225
EOLN	227
<i>Pitfall—Use of <code>eo ln</code> with Numeric Data</i>	228
Reading from a File (Optional)	228
Off-Line Data and a Preview of EOF (Optional)	229
Modifying an Algorithm	229
The Repeat Statement	230
Comparison of the <i>while</i> and Repeat Loops	231
<b>TURBO Pascal—<code>str</code> and <code>val</code> (Optional)</b>	232
Self-Test Exercises	233
<i>Case Study—Testing a Procedure</i>	233
<i>Case Study—Finding the Largest and Smallest Values on a List</i>	235
Unrolling a Loop	237

Designing Robust Programs	239
Invariant Assertions and Variants Expressions (Optional)	240
<i>for</i> Statements	244
Example—Summing a Series	247
Repeat N Times	248
Invariant Assertions and <i>for</i> Loops (Optional)	249
What Kind of Loop to Use	249
Nested Loops	250
Debugging Loops	250
<i>Case Study</i> —Calendar Display	254
Starting Over	258
Summary of Problem Solving and Programming Techniques	259
Summary of Pascal Constructs	259
Exercises	262
References for Further Reading	265

## CHAPTER EIGHT

### Designing Functions and Data Types 267

Use of Functions	269
A Sample Function Declaration	269
<i>Case Study</i> —A Function to Compute Powers	271
<i>Pitfall</i> —Thinking the Function Name is a Variable	272
<i>Pitfall</i> —Special Cases	274
Local Identifiers	274
Functions That Change Their Minds	277
Side Effects	278
Self-Test Exercises	279
Boolean-Valued Functions	280
<i>Case Study</i> —Testing for Primes (Optional)	280
More Standard Functions (Optional)	284
Random Number Generators (Optional)	284
Designing Your Own Pseudorandom Number Generator (Optional)	286
A Better Method for Scaling Random Numbers (Optional)	289
<b>TURBO Pascal</b> —A Predefined Random Number Generator (Optional)	290
Self-Test Exercises	291
Ordinal Types	293
<b>TURBO Pascal</b> —String Comparisons	294
The Functions <i>pred</i> , <i>succ</i> , <i>ord</i> , and <i>chr</i> (Optional)	295
<b>TURBO Pascal</b> — <i>upcase</i>	296
Subrange Types	296
<b>TURBO Pascal</b> —The <i>R</i> Compiler Directive	299
The <i>for</i> and <i>case</i> Statements Revisited	299
Allowable Parameter Types	300
<b>TURBO Pascal</b> —String Parameters	300
<i>Pitfall</i> —Parameter Type Conflicts	301



<i>Pitfall</i> —Anonymous Types	302
<i>Case Study</i> —Complete Calendar Program	303
Use of Subrange Types for Error Detection	307
Enumerated Types (Optional)	310
TURBO Pascal—String Functions and Procedures	313
<i>TURBO Pascal Case Study</i> —Replacing Substrings	315
Summary of Problem Solving and Programming Techniques	320
TURBO Pascal—Summary of String Functions and Procedures	321
Summary of Pascal Constructs	324
Exercises	325
References for Further Reading	330

## CHAPTER NINE

**Arrays for Problem Solving****331**

Introduction to Arrays	333
<i>Pitfall</i> —Use of Plurals in Array Definitions	338
Type Declarations—A Summary	338
Input and Output with Arrays	339
Partially Filled Arrays	340
<i>Pitfall</i> —Array Index Out of Range	341
TURBO Pascal—More about the R Compiler Directive	343
Self-Test Exercises	344
The Notion of a Data Type	345
Arrays as a Structured Type	346
Allowable Function Types	348
<i>Case Study</i> —Searching an Array	348
<i>Pitfall</i> —Type Mismatches with Array Parameters	352
Efficiency of Variable Parameters	352
Array Example with Noninteger Indexes	354
Random versus Sequential Array Access	355
Array Indexes with Semantic Content	355
Enumerated Types as Array Index Types (Optional)	355
<i>Case Study</i> —Production Graph	357
Off-Line Data (Optional)	362
<i>Case Study</i> —Sorting	362
Summary of Problem Solving and Programming Techniques	364
Summary of Pascal Constructs	367
Exercises	369
References for Further Reading	371

## CHAPTER TEN

**Complex Array Structures****373**

Strings of Characters	375
Arrays of Arrays	375



<i>TURBO Pascal—The Types <code>array of char</code> and <code>string</code></i>	379
Parallel Arrays	379
<i>TURBO Pascal Case Study—Making a History Table</i>	380
Self-Test Exercises	383
The Notion of a Data Structure	387
Data Abstraction	388
<i>TURBO Pascal—Clearing the Screen</i>	389
<i>TURBO Pascal Case Study—Automated Drill</i>	389
Adapting a Known Algorithm	394
<i>Standard Pascal Case Study—Pattern Matching (Optional)</i>	394
Design by Concrete Example	399
Standard Pascal—Packed Arrays (Optional)	404
Standard Pascal—Packed Arrays of Characters (Optional)	405
Self-Test Exercises	407
Multidimensional Arrays	407
<i>Case Study—Grading Program</i>	410
<i>Pitfall—Exceeding Storage Capacity</i>	411
<i>TURBO Pascal—Typed Constants (Optional)</i>	412
<i>TURBO Pascal—Array Constants (Optional)</i>	415
Summary of Problem Solving and Programming Techniques	416
Summary of Pascal Constructs	417
Exercises	418
References for Further Reading	423

## CHAPTER ELEVEN

**Records and Other Data Structures****425**

Introduction to Records	427
Comparison of Arrays and Records	431
The Syntax of Simple Records	432
Self-Test Exercises	433
Records within Records	434
Arrays of Records	434
Sample Program Using Records	436
Choosing a Data Structure	438
The <code>with</code> Statement	439
<i>Pitfall—Problems with the <code>with</code> Statement</i>	440
<i>TURBO Pascal—Use of Strings in Records</i>	440
<i>TURBO Pascal Case Study—Sales Report</i>	441
<i>Case Study—Strings Implemented as Records (Optional)</i>	442
<i>Case Study—Sorting Records</i>	449
Searching by Hashing (Optional)	451
Variant Records (Optional)	455
Simple Uses of Sets	459
<i>TURBO Pascal Pitfall—Limitations on Sets</i>	464
More about Sets (Optional)	464

TURBO Pascal—Defined Set Constants (Optional)	468
Summary of Problem Solving and Programming Techniques	469
Summary of Pascal Constructs	470
Exercises	473
References for Further Reading	476

## CHAPTER TWELVE

**Program Design Methodology** 479

Some Guidelines for Designing Algorithms	481
Writing Code	484
Data and Procedural Abstraction	484
Testing and Debugging	485
Verification	491
Portability	492
Efficiency	492
Efficiency versus Clarity	495
Off-Line Data	496
Coping with Large Programming Tasks	496
Summary of Terms	497
Exercises	499
References for Further Reading	501

## CHAPTER THIRTEEN

**Text Files and Secondary Storage** 503

Part I	505
Text Files	505
Writing and Reading Text Files	506
TURBO Pascal—Opening Files	507
TURBO Pascal—More about File Names (Optional)	511
Standard Pascal—File Handling (Optional)	512
<i>Pitfall</i> —Mixing Reading and Writing to a Text File	513
<i>Pitfall</i> —The Silent Program	514
Self-Test Exercise	514
Interactive Exercises	515
Part II	516
read and write Reexamined	516
eof and eoln	518
Using a Buffer	522
<i>Pitfall</i> —Forgetting the File Variable in eof or eoln	522
<i>Pitfall</i> —Use of eoln and eof with Numeric Data	523
Text Files as Parameters to Procedures	523
<i>Pitfall</i> —Portability	525
Basic Technique for Editing Text Files	525

TURBO Pascal—Temporary Files	525
<i>TURBO Pascal Case Study—Editing Out Excess Blanks</i>	526
Text Editing as a Programming Aid	529
Summary of Problem Solving and Programming Techniques	531
Summary of Pascal Constructs	532
Exercises	536

## CHAPTER FOURTEEN

### **Problem Solving Using Recursion** **541**

<i>Case Study—A Recursive Function</i>	543
A Closer Look at Recursion	545
<i>Pitfall—Infinite Recursion</i>	547
Stacks	550
<i>Pitfalls—Stack Overflow</i>	551
Self-Test Exercises	554
Proving Termination and Correctness for Recursive Functions (Optional)	555
<i>Case Study—A Simple Example of a Recursive Procedure</i>	557
Technique for Designing Recursive Algorithms	560
<i>Case Study—Towers of Hanoi—An Example of Recursive Thinking</i>	560
Recursive versus Iterative Procedures and Functions	565
<i>Case Study—Binary Search</i>	567
Forward Declarations (Optional)	572
Summary of Problem Solving and Programming Techniques	574
Exercises	575
References for Further Reading	577

## CHAPTER FIFTEEN

### **Solving Numeric Problems** **579**

A Hypothetical Decimal Computer	581
Binary Numerals (Optional)	585
Machine Representation of Numbers in Binary (Optional)	586
Extra Precision (Optional)	588
Self-Test Exercises	589
<i>Pitfall—Sources of Error in Real Arithmetic</i>	589
<i>Pitfall—Error Propagation</i>	592
<i>Case Study—Series Evaluation</i>	593
<i>Case Study—Finding a Root of a Function</i>	594
Summary of Problem Solving and Programming Techniques	600
Summary of Terms	600
Exercises	600
References for Further Reading	603

## CHAPTER SIXTEEN

**More File Types****605**

The General Notion of a File	607
File Variables	608
<i>TURBO Pascal—Opening Files</i>	608
Standard Pascal—Opening Files (Optional)	609
Windows	609
read and write	610
<i>TURBO Pascal Case Study—Processing a File of Numeric Data</i>	611
<i>Pitfall—Unexpected End of File</i>	612
Self-Test Exercises	613
<i>TURBO Pascal Example—Creating a File of Records</i>	615
Deciding What Type of File to Use	616
<i>TURBO Pascal—seek</i>	616
<i>TURBO Pascal Examples—Random and Sequential File Access</i>	618
Files as Parameters to Procedures	622
<i>TURBO Pascal Case Study—Merging Two Files</i>	622
Summary of Problem Solving and Programming Techniques	629
Summary of Pascal Constructs	629
Exercises	632

## CHAPTER SEVENTEEN

**Dynamic Data Structures****635**

The Notion of a Pointer	637
Pascal Pointers and Dynamic Variables	637
Manipulating Pointers	640
Nodes	643
The Pointer <i>nil</i>	644
<i>Pitfall—Forgetting that nil Is a Pointer</i>	644
Linked Lists—An Example of Pointer Use	645
<i>Case Study—Building a Linked List</i>	646
The Empty List	650
<i>Pitfall—Losing Nodes</i>	650
<i>Case Study—Tools for Manipulating a Linked List</i>	651
Self-Test Exercises	660
Pointer Functions	661
<i>Pitfall—Testing for the Empty List</i>	662
<i>Case Study—Using a Linked List to Sort a File</i>	663
Stacks	667
dispose (Optional)	670
Implementation (Optional)	671
<i>TURBO Pascal—mark and release (Optional)</i>	675
Doubly Linked Lists—A Variation on Simple Linked Lists	676



Trees	678
<i>Case Study—Building a Search Tree</i>	683
Summary of Problem Solving and Programming Techniques	685
Summary of Pascal Constructs	686
Exercises	688
References for Further Reading	690

## CHAPTER EIGHTEEN

**Units for Modularization and Abstraction****691****Part I** 693

Introduction to Units	693
Writing a Simple Unit	693
Predefined Units	696
Interactive Exercises	696
<i>Version 4.0—Using Multiple Units</i>	696
<i>Version 5.0—Using Multiple Units</i>	697
Interactive Exercises	698
Units and Procedural Abstraction	698
Owned Variables and the Initialization Section	700
<i>Version 5.0—Uses Clause in Implementation Section (Optional)</i>	703
Unit Directories (Optional)	703

**Part II** 705

Units and Data Abstraction	705
<i>Case Study—A Unit for a Queue</i>	706
Summary of Problem Solving and Programming Techniques	710
Summary of TURBO Pascal Units	711
Programming Exercises	711

## APPENDIX ONE

The <i>goto</i> and <i>exit</i> Statements	A.1
--	-----

## APPENDIX TWO

<i>TURBO Pascal—More Integer Types</i>	A.5
--	-----

## APPENDIX THREE

Precedence of Operators	A.6
-------------------------	-----

## APPENDIX FOUR

<i>Syntax Diagrams for TURBO Pascal Programs</i>	A.7
--	-----

## APPENDIX FIVE

<i>A Brief Introduction to the IBM PC and PC-DOS</i>	A.22
--	------

## APPENDIX SIX

<i>Summary of DOS Commands</i>	A.34
--------------------------------	------

## APPENDIX SEVEN

<i>A Quick Introduction to the TURBO Menu Environment, with Emphasis on the Editor</i>	A.36
--	------

## APPENDIX EIGHT

The TURBO File Menu      A.49

## APPENDIX NINE

TURBO Hotkeys      A.53

## APPENDIX TEN

The TURBO Editor Commands      A.55

## APPENDIX ELEVEN

Compiling to Disk in TURBO      A.58

## APPENDIX TWELVE

Input/Output Redirection in DOS      A.60

## APPENDIX THIRTEEN

The TURBO Crt Unit      A.62

## APPENDIX FOURTEEN

TURBO Pascal—Character Graphics      A.66

## APPENDIX FIFTEEN

TURBO Pascal—Text Windows      A.69

## APPENDIX SIXTEEN

Some Common TURBO Pascal Compiler Directives      A.72

## APPENDIX SEVENTEEN

TURBO Pascal—I/O Error Handling      A.75

## APPENDIX EIGHTEEN

Version 5.0-The Integrated Debugger      A.77

## APPENDIX NINETEEN

ASCII Character Set      A.86

Glossary      A.88

Answers to Self-Test Exercises      A.97

Index      I.1

## Inside Back Cover

Reserved Words

Ordinal Types

Types that May Be Used for the Value Returned by a Function

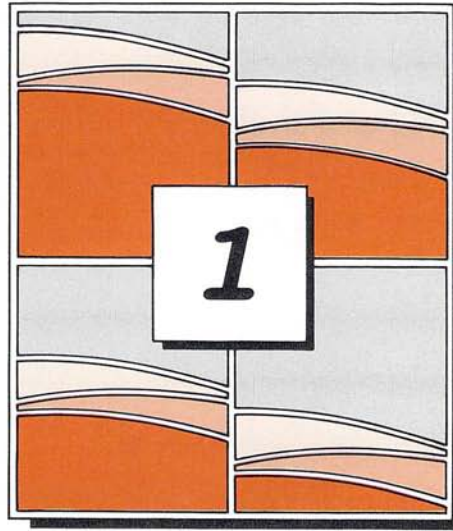
Ranges for TURBO Pascal Numbers

---









# *Introduction to Computers and Problem Solving*

The arithmetical machine produces effects which  
approach nearer to thought than all the actions of animals.  
But it does nothing which would enable us  
to attribute will to it, as to the animals.

*Blaise Pascal*

## Chapter Contents

**What a Computer Is**  
**The Modern Digital Computer**  
**The Notion of an Algorithm**  
**Programs and Data**  
**High Level Languages**  
**The Pascal Language**

**Programming Environments**  
**Designing Programs**  
**Summary of Terms**  
**Exercises**  
**References for Further Reading**

**I**n this chapter we outline some of the basic concepts common to all computer systems and all programming languages. The theme here and throughout this book is that there is a methodology of effective programming that is relatively independent of the particular programming language used or the particular computer used. In fact, this chapter presents no details of the Pascal language.

---

## What a Computer Is

In less than one human lifetime, computers have evolved from a scientific curiosity to an indispensable tool in virtually all areas of our lives. They handle our financial transactions, control manufacturing processes, keep track of airline reservations, forecast weather, control space probes—the list goes on seemingly without end. But just what are these things called *computers*?

The basic nature of computers is surprisingly simple. Computers are machines that perform very simple tasks according to specific instructions. Their ability to perform so many simple tasks at such great speed and with such a high degree of accuracy is what makes computers so useful. One can think of a computer as a clerk who does nothing all day but sit and perform trivial, routine tasks according to given sets of instructions, and who does so with perfect accuracy, infinite patience, a flawless memory, and unimaginable speed.

---

## The Modern Digital Computer

A set of instructions for a computer to follow is called a *program*. The collection of programs used by a computer is referred to as the *software* for that computer. The physical machines that make up a computer installation are referred to as *hardware*. In this book we are concerned almost exclusively with software, but a brief overview of how the hardware is organized will be useful.

*software/  
hardware*

Most computers are organized as shown in Figure 1.1. They can be thought of as having four main components: the input device(s), the output device(s), the central processing unit (CPU), and the memory.

An *input device* is any device that allows a person to communicate information to the computer. For readers of this book, the input device is likely to be a keyboard rather like a typewriter keyboard, but it could instead be some other type of device, or it could consist of a variety of devices.

*input  
keyboard*

An *output device* performs the opposite task. It allows the computer to communicate information to the user. One of the most common output devices is a *display screen* that resembles a television screen. This display screen is often referred to as a *CRT screen* or *monitor*. (The initials CRT stand for “cathode ray tube”.) Often there is more than one output device. For example, in addition to the display screen there may be a typewriter or typewriterlike device to produce printed output. These devices for producing printed output are called, appropriately enough, *printers*. The keyboard and display screen are frequently thought of as a single unit called a *video display terminal* or simply a *terminal*.

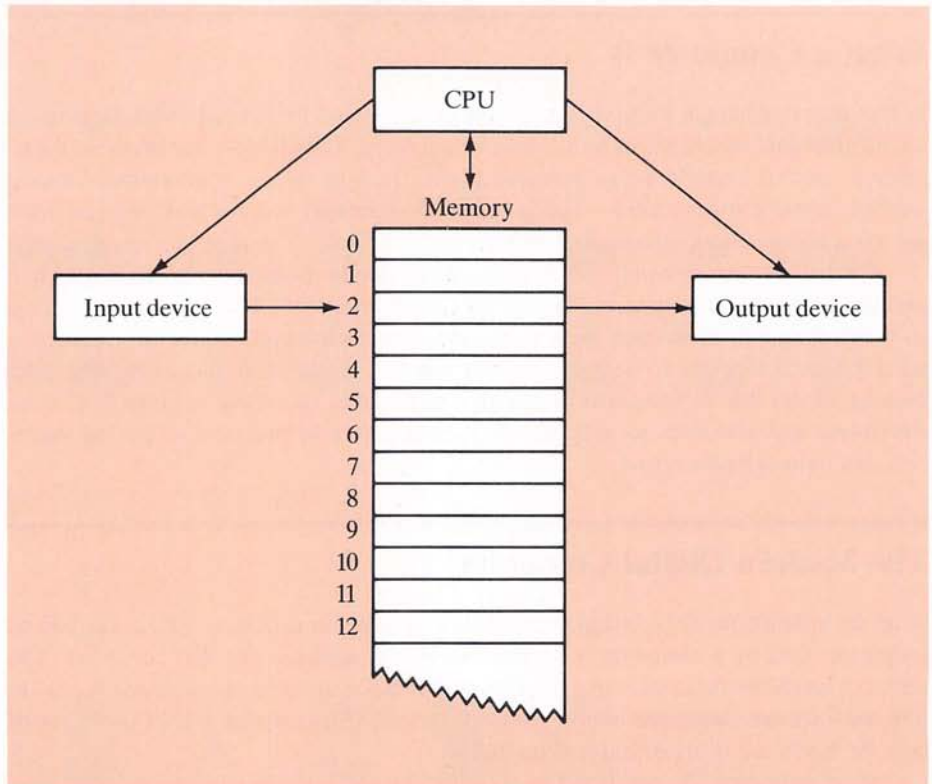
*output  
display screen*

In order to store input and to have the equivalent of scratch paper for performing calculations, computers are provided with *memory*. The memory is very simple. It consists of a long list of numbered locations called *words* or, more descriptively, *memory locations*. The number of memory locations varies from one computer to another, ranging from a few thousand to many millions. In fact, memory may be added to a computer almost without limit, although once the memory size exceeds certain thresholds, the computer system must be made a bit more sophisticated. Each memory loca-

*memory*

---





**Figure 1.1**  
**Main components**  
**of a computer.**

tion or word contains a string of zeros and ones. The contents of these locations can change. Hence, you can think of each memory location as a tiny blackboard that the computer may write on and then erase. In most computers, all locations contain the same number of zero/one digits; some typical sizes are 16, 32, and 64 digits. A digit that assumes only the values zero or one is called a *bit*. Hence, if you are working on a 32-bit machine, it means that each memory location in your computer can hold 32 bits, that is, 32 digits, each either zero or one.

That the information in a computer's memory is represented as zeros and ones need not be of great concern to a person programming in Pascal. The reasons for using only zeros and ones have to do with the physics of hardware design. Computers using larger repertoires of digits can and have been designed. The use of zeros and ones does, however, have a few implications that you should be aware of. First, the computer has to do its arithmetic in what is known as "binary notation." (We will discuss binary arithmetic in Chapter 15.) A more important point is that the computer needs to interpret these strings of zeros and ones as numbers or letters or instructions or other types of information. The computer performs these interpretations automatically according to certain codes. A different code is used for each type of item that can be stored in a location: There is one code for letters, another for whole numbers, another for fractions, another for instructions, and so on. For example, in one commonly used set of codes, 1000001 is the code for both the letter A and the number 65. In order to know

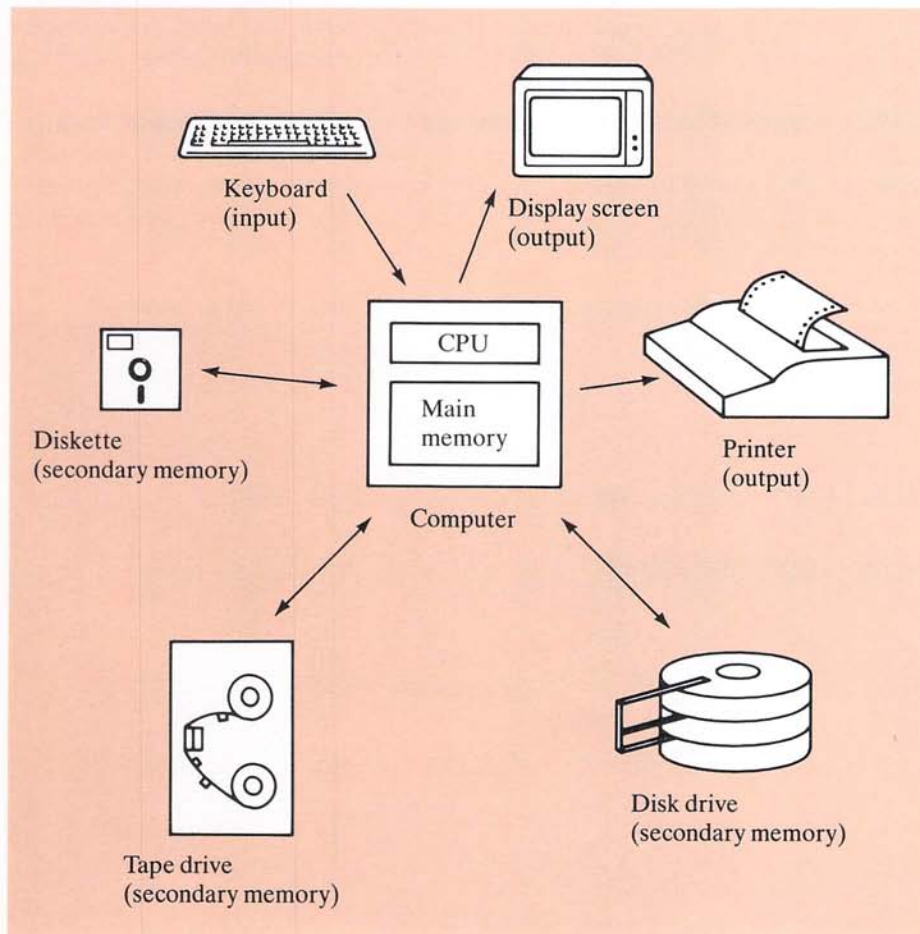


what the string 1000001 means in a particular location, the computer must keep track of which code is currently being used for that location. Fortunately for us, the programmer seldom needs to be concerned with such codes and can safely proceed as though the locations actually contained letters, numbers, or whatever type of information is desired.

The memory we just described is called *main memory*. Most computers have additional memory called *secondary memory*, also frequently called *secondary storage* or *auxiliary storage*. Main memory serves as a temporary memory that is used only while the computer is actually following the instructions in a program. Secondary memory is used for keeping a permanent record of information after (and before) the computer is used. On small computers secondary memory is likely to consist of something called a *floppy disk* or *diskette*, and on larger computers it is likely to be something called a *hard disk*. Magnetic tape units are also commonly used for secondary memory. A typical computer installation with different kinds of memory is depicted in Figure 1.2. We will not be concerned with secondary memory until we reach Chapters 13 and 16.

*secondary  
memory*

*disks*



**Figure 1.2**  
A typical computer  
installation.

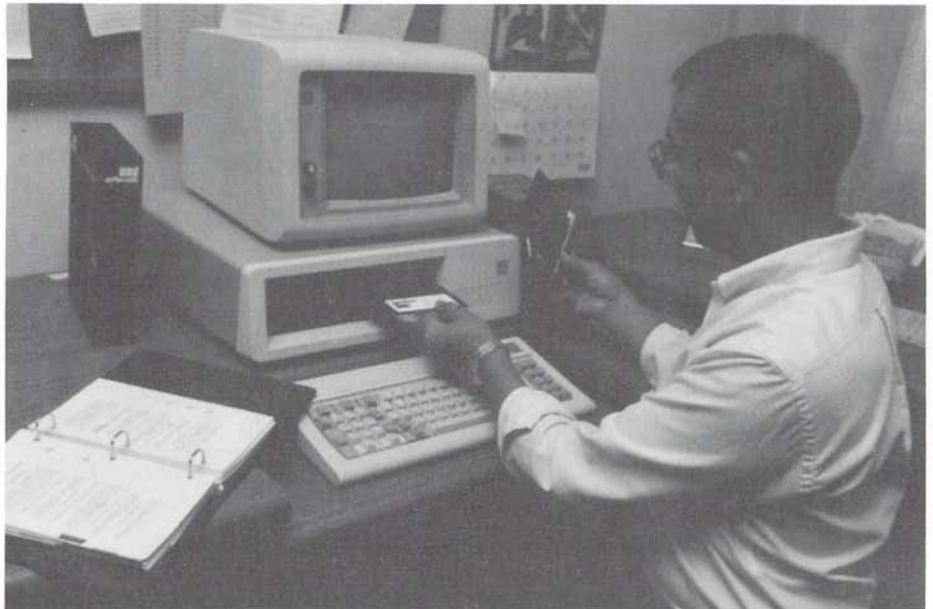
**CPU**

The *central processing unit*, or *CPU* for short, is the “brain” of the computer. It is the CPU that follows the instructions in a program and performs the calculations specified by the program. The CPU is, however, a very simple brain. All that it can do is follow a set of simple instructions provided to it by the programmer.

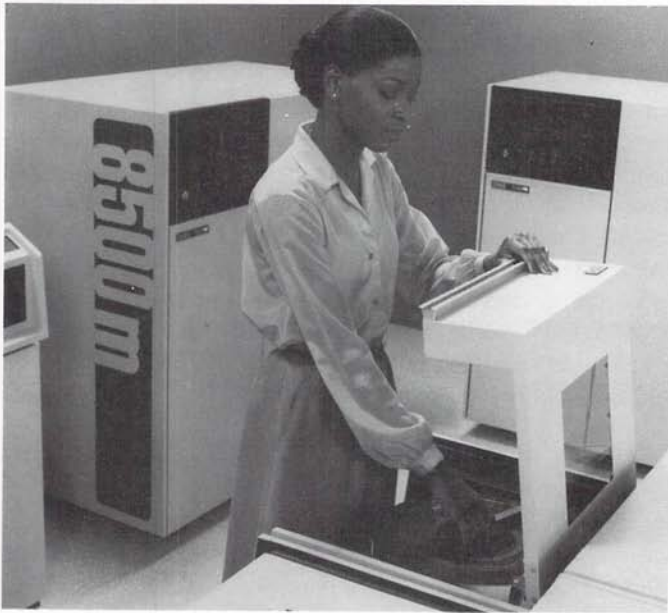
Typical CPU instructions say things like “interpret the zeros and ones as numbers and then add the number in memory location 37 to the number in memory location 59 and put the answer in location 83” or “read a letter of input, convert it to its code as a string of zeros and ones, and place it in memory location 1298.” The CPU can perform subtraction, multiplication, and division as well as addition. It can move things from one memory location to another. It can interpret strings of zeros and ones as letters and send the letters to an output device. The CPU also has some primitive ability to rearrange the order of instructions. Needless to say, CPU instructions vary somewhat from computer to computer. The CPU of a modern computer can have as many as several hundred available instructions. However, these instructions are typically all about as simple as those just described.

At this point you may be wondering where in the computer the program is kept. The answer is: in the memory. Thus, the memory serves both as a place to store the program and as a kind of “scratch paper” for doing calculations. Usually, we conceptualize the program as being outside of memory, but occasionally we will need to be aware of the fact that it actually resides in memory.

That is it. That is all there is to a computer. Conceptually, it is a simple machine. Its power comes from the size of its memory, its speed, its accuracy, and the sophistication of its programs. Your computer may not be configured exactly as we have portrayed it, but it will be similar and, more important, will behave exactly as if it were the very machine we have just described.



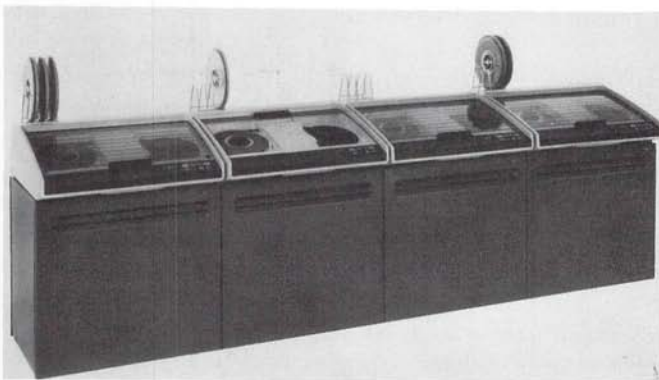
**A floppy disk being inserted into a disk drive.**



**Hard disk pack  
being inserted into  
a disk drive.**



**Hard disk pack.**



**Tape drives.**



## The Notion of an Algorithm

When learning your first programming language, it is easy to get the impression that the hardest part of solving a problem on a computer is translating your ideas into the specific language that will be fed into the computer. This definitely is not the case. The most difficult part of solving a problem on a computer is coming up with the method of solution. After you have developed a method of solution, it is routine to translate your method into the required language, be it Pascal or some other programming language. When solving a problem with a computer, it is therefore helpful to temporarily ignore the computer programming language and to concentrate instead on formulating the steps of the solution and writing them down in plain English, as if the instructions were to be given to a human being. A set of instructions expressed in this way is frequently referred to as an “algorithm.”

*Algorithm*

A set of instructions that leads to a solution is called an *algorithm*. Some approximately equivalent words are “recipe,” “method,” “directions,” and “routine.” The instructions may be expressed in a programming language or a human language. Our algorithms will be expressed in English and in the programming language Pascal. An algorithm expressed in a language that a computer can understand is called a *program*, which explains why computer languages are called *programming languages*.

*program*

The word “algorithm” has a long history, but its meaning has recently taken on a new character. The word itself derives from the name of the ninth-century Arabic mathematician and astronomer Al-Khowarizmi, who wrote an early and famous textbook on the manipulation of numbers and equations entitled *Kitab al-jabr w'al-muqabala*. The similar-sounding word “algebra” was derived from the Arabic word “al-jabr,” which appears in the title of the text and is often translated as “reuniting” or “restoring.” The entire title can be translated as “Rules for Reuniting and Reducing.” The meanings of the words “algebra” and “algorithm” used to be much more intimately related than they are now. Indeed, until very recently the word “algorithm” usually referred to algebraic rules for solving numeric equations.

Today the word “algorithm” refers to a wide variety of instructions for manipulating symbolic as well as numeric entities. The properties that qualify a set of instructions as an algorithm now are determined by the nature of the instructions and not by the things to which they apply. To qualify as an algorithm, a set of instructions must completely and unambiguously specify the steps to be taken and the order in which they are to be performed. The instructions cannot rely on any intelligence on the part of the person or machine following the instructions. The follower of an algorithm does exactly what the algorithm says, neither more nor less.

*sample  
Algorithm*

An example may help to clarify the concept. Figure 1.3 contains an algorithm expressed in rather stylized English. The algorithm determines the number of times a specified name occurs on a list of names. If the list contains the winners of each of last season’s football games and the name is that of your favorite team, then the algorithm determines how many games your team won. The algorithm is short and simple but is otherwise typical of the algorithms we will be dealing with.

The instructions numbered 1 through 5 in our sample algorithm are meant to be carried out in that order. Unless otherwise specified, we will always assume that the

---



```
begin
1. Request the list of names and call it NameList;
2. Request the name being sought and call it KeyName;
3. On a blackboard called Count write the number zero;
4. repeat the following for each name on NameList:
    if the name is the same as KeyName
    then add one to the number written on Count;
    {the old number is erased, leaving only one number
    on Count}
5. Announce the number written on Count as the answer.
end.
```

**Figure 1.3**  
**An algorithm.**

instructions of an algorithm are carried out in the order in which they are listed. Most interesting algorithms do, however, specify some change of order, usually a repetition of some instruction again and again, as in instruction 4 of our sample algorithm.

This simple example illustrates a number of important points about algorithms. Algorithms are usually given some information. In our example, the algorithm was given a name and a list of names. The information that is given to an algorithm is called *input* or *data*. Algorithms usually give an answer, or answers, back. In the example, the answer was a number. The answers given by an algorithm are called *output*. In addition to being able to remember input and output, algorithms typically need to remember some other information. In the example, a single number was remembered; the number changed as the algorithm proceeded, and only the last value of the number was output.

One final observation about our sample algorithm: It always ends. No matter how long the list is, the algorithm always gets to the end of the list and announces an answer. There are algorithms that never terminate. Common examples are the algorithms used by computerized airline reservation systems. They never terminate; they just keep adding and deleting reservations forever, or until the airline goes bankrupt or changes its computer system. An algorithm that might not end is called a *partial algorithm*. An algorithm that is guaranteed to end is called a *total algorithm*. Some authors, especially in more advanced texts, reserve the word “algorithm” for what we called total algorithms. However, we will use the word to mean any algorithm, whether or not it is guaranteed to terminate.

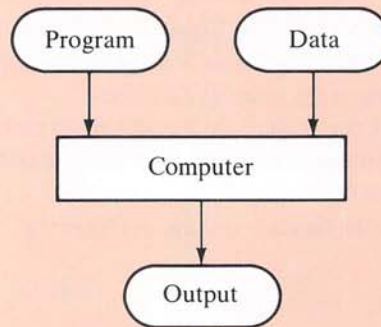
*input or  
data*

*output*

---

## Programs and Data

As we have already noted, a program is just an algorithm written in a language that can be fed into a computer. As shown in Figure 1.4, the input to a computer can be thought of as consisting of two parts: a program and some data. The data is what we conceptualize as the input to the algorithm that the computer will follow. In other words, the data is the input to the program, and both the program and the data are input to the computer. The word “input” is thus being used in two slightly different ways. This does



**Figure 1.4**  
Simple view of  
running a  
program.

*running  
a program*

require some care to keep from getting confused, but it is standard usage and you may as well get used to it. For the sample algorithm in the previous section, the data was the name sought and the list of names to be searched. In order to get a computer to carry out the algorithm, both the algorithm (translated into a programming language) and the data are given as input to the computer. Whenever we give both a program and some data to a computer, we are said to be *running* the program on the data, and the computer is said to be *executing* the program on the data.

The word *data* also has a much more general meaning than the one we have just given it. In its most general sense, it means any information available to the computer or to some part of the computer. The word is commonly used in both the narrow sense and the more general sense. One must rely on context to decide which meaning is intended.

---

## High Level Languages

*high level  
language*

The language Pascal is a *high level language*, as are most of the other programming languages you are likely to have heard of, such as FORTRAN, BASIC, COBOL, C, Modula, and Ada. High level languages resemble human language in many ways. They are designed to be easy for human beings to write programs in and easy for human beings to read. Like most high level languages, Pascal uses English words combined in ways that resemble English sentences. For example, the following is a line from a Pascal program:

```
if (X = Y) and (Z = W) then  
    write('the answer is 42')
```

You can read and understand this instruction almost without any explanation.

---

A high level language, like Pascal, contains instructions that are much more complicated than the simple instructions a computer's CPU is capable of following. The kind of language a computer can understand is called a *low level language*. A typical low level instruction might be the following:

ADD X Y Z

This instruction might mean "add the number in the memory location called X to the number in the memory location called Y and place the result in the memory location called Z."

The above sample instruction is written in what is called *assembly language*. Although assembly language is almost the same as the language understood by the computer, it must still undergo one simple translation before the computer can understand it. For a computer to follow an assembly language instruction, the words need to be translated into strings of zeros and ones. For example, ADD might translate to 0110, the X might translate to 1001, the Y to 1010, and the Z to 1011. The version of the instruction that the computer ultimately follows would then be

0110100110101011

Programs written in the form of zeros and ones are said to be written in *machine language*, because that is the version of the program that the computer (the "machine") actually reads and follows. Assembly language and machine language are almost the same thing, and the distinction between them will not be important to us. The important distinction is that between machine language and a high level language such as Pascal.

Do not bother to memorize our assembly language instruction to add two numbers, nor its translation into a string of zeros and ones. The exact assembly language instructions and their translation into zeros and ones will differ from machine to machine. The only point to remember is that any high level language must be translated into machine language before the computer can understand and follow the program.

A program that translates a high level language, like Pascal, to a machine language is called a *compiler*. A compiler is thus a somewhat peculiar sort of program in that its input or data is some other program and its output is yet another program. To avoid confusion, the input program is usually called the *source program* and the translated version is called the *object program* or *object code*. The word *code* is frequently used to mean a program or a part of a program, and this usage is particularly common when referring to object programs.

Now, suppose you want to run a Pascal program. In order to get the computer to follow your Pascal instructions, proceed as follows. First, run the compiler, using your Pascal program as data. Notice that in this case the Pascal program is not being treated as a set of instructions. To the compiler your Pascal program is just a long string of characters. The output will be another long string of characters, which is the machine language equivalent of the Pascal program. Next, run this machine language program on what we normally think of as the data for the Pascal program. The output will be what we normally conceptualize as the output of the Pascal program. The process is easier to visualize if you have two computers available, as diagrammed in Figure 1.5.

In reality, the entire process just described is facilitated on one computer by special

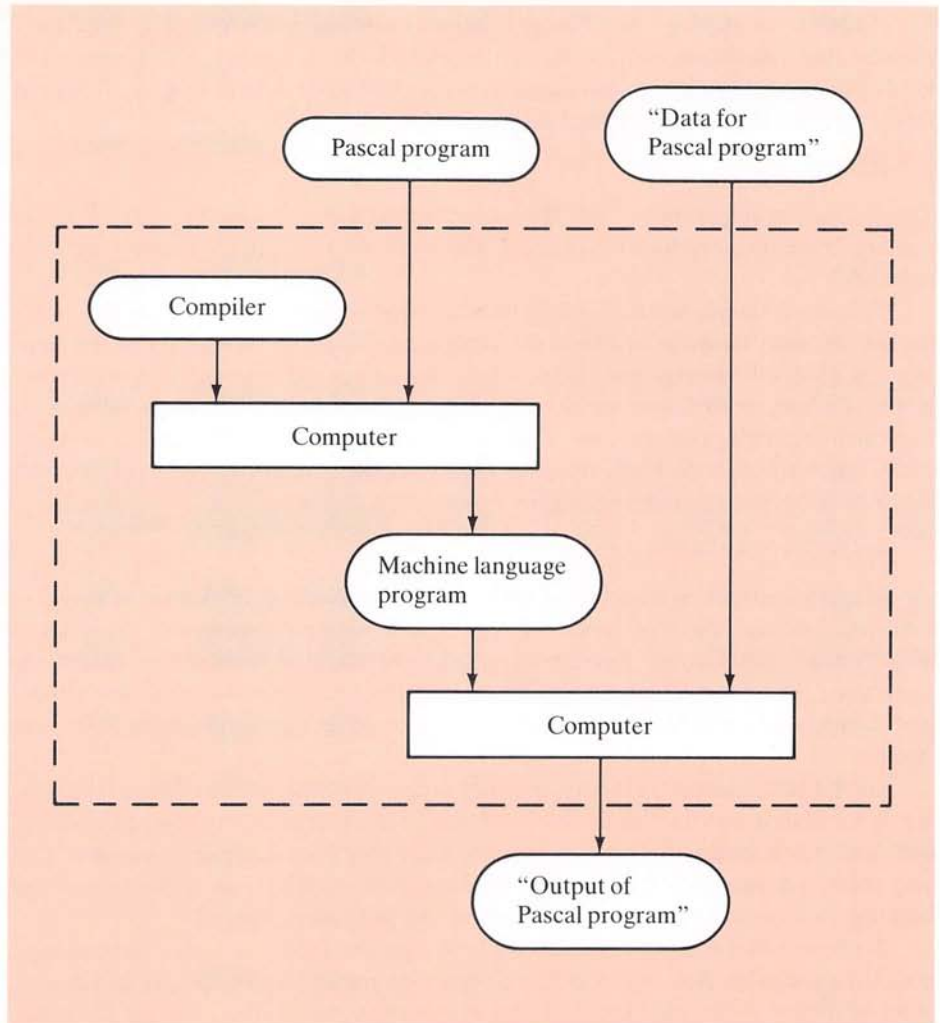
*low level  
language*

*assembly  
language*

*machine  
language*

*compiler*





**Figure 1.5**  
Compiling and  
running a Pascal  
program.

programs called *systems programs*. Although there is but one computer, the systems programs make it appear as though a big box, represented by the dotted line in Figure 1.5, were built around two computers. You simply place your Pascal program and data in the box and the rest is taken care of automatically. Hence, you can think of the computer as actually running the Pascal program. None of this is peculiar to Pascal. The translation process is the same with any high level programming language.

---

## The Pascal Language

In this book we will use a programming language called *Pascal*. Pascal is a high level, general-purpose programming language. When we say Pascal is a *general-purpose language*, we mean that it is suitable for a diverse range of applications. Indeed, it is com-

---



monly used to write programs for a wide variety of applications, including programs for numeric scientific calculations, for business data processing, and for text editing, as well as to write various systems programs, including compilers.

The Pascal language was developed by Professor Niklaus Wirth and his colleagues at the Eidgenössische Technische Hochschule in Zurich, Switzerland, during the late 1960s and early 1970s. Wirth designed Pascal to be a good first language for people learning to program. As such, Pascal has relatively few concepts to learn and digest. It has a design that facilitates the writing of programs in a style that is now generally accepted as good standard programming practice. Another of Wirth's design goals was ease of implementation. He designed the language so that it is relatively easy to write a Pascal compiler for a new type of computer. This is one of the reasons that Pascal has become available on so many different computers in such a short amount of time. Needless to say, Pascal does have its shortcomings, and they will become apparent as you learn the language. Still, it is one of the best languages to use when learning to program computers. Moreover, it does not lose its usefulness after you learn the rudiments of programming. It is widely available and is frequently used by professional programmers. Also, a number of other popular programming languages are similar to Pascal and are thus easy to learn once you have mastered Pascal.

---

## Programming Environments

Our description of compiling and running a Pascal program is a correct but oversimplified picture of what actually happens. Systems programs on modern computers include many other programs besides the compiler. The main systems program is called the *operating system*. It is the program in charge of other programs—the “manager,” so to speak. It keeps track of which program is running on which piece of data; it brings out the editor program, which allows you to use the computer as a typewriter to write up your programs; it puts the editor program away and brings out the appropriate compiler when needed; and it also runs the machine language program that the compiler produces. On a computer with many simultaneous users, the operating system also keeps the various users from interfering with one another. It does this by moving resources from one user to another at such high speeds that, unless the number of users is very high, each person is given the illusion of being the only one using the computer. The operating system also does whatever accounting and security checking is needed. While all this is going on, the programs that are running usually produce various pieces of information as output. For example, the compiler will look for mistakes in your program and will give one or more output messages should it find any errors.

*operating  
system*

Computer facilities are now configured in a wide variety of ways. Smaller computers called *microcomputers* or *personal computers* are dedicated to a single user. Larger computers called *mainframes* can serve numerous users simultaneously by means of special operating systems called *time-sharing systems*. Often, the computer facility will consist of a *network* connecting a number of different computers so that they can share certain resources such as printers and secondary storage devices. The particular configuration you are working on will matter little to our study of programming techniques. For our purposes they all serve the same function and behave similarly.

---

*editor**files*

We have outlined the basic tasks involved in running a program. They are common to all systems. The details will vary from system to system, however, and you must find out many of these details before you can run a program. Your system will have a program called an *editor* that lets you use the computer as a typewriter. The system will also allow you to store programs and to retrieve them at a later time. The program that controls this storing and retrieving is usually called a *file manager*, or something similar, such as *file system* or *filer*. You will need to learn how to use both the editor and the file manager. Finally, you will have to learn how to get the compiler to translate your program into machine code and how to run the machine language program. (On many systems the processes of compiling and then running the program are combined into a single process.)

Although you will need to find out these details about your computer system, at this point you do not need to know anything about how to write a program in Pascal or in any other programming language. The rest of this book is devoted to teaching you how to go from a problem to a Pascal program to solve that problem.

---

## Designing Programs

Designing a program is frequently a difficult task. There is no complete set of rules, no algorithm to tell you how to write programs. Program design is a creative process. Still, there is the outline of a plan to follow. The outline is given in diagrammatic form in Figure 1.6. As indicated there, the program-design process can be divided into two phases: the problem solving phase and the implementation phase. The result of the *problem solving phase* is an algorithm for solving the problem. The algorithm is expressed in English. To produce a program in a programming language such as Pascal, the algorithm is translated into the programming language. Producing the final program from the algorithm is called the *implementation phase*.

*problem definition*

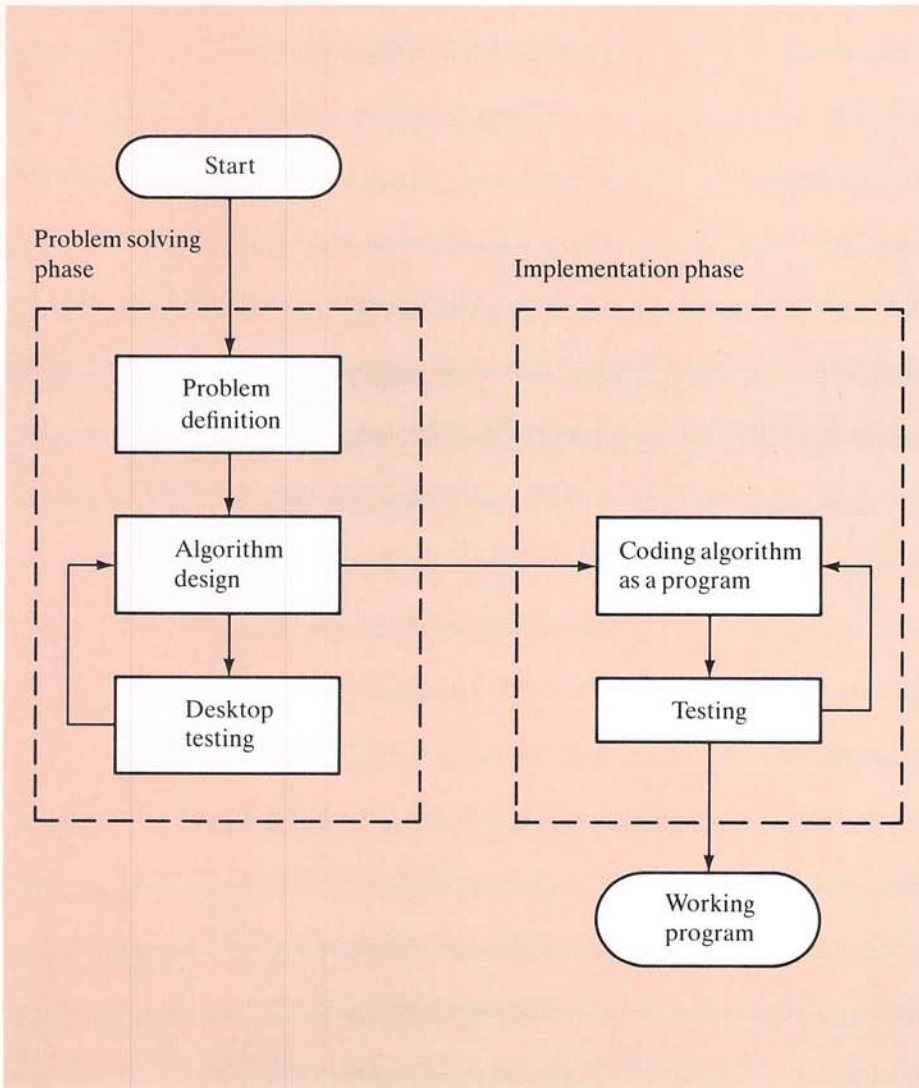
The first step, in both the problem solving phase and the entire design process, is to be certain that the task is completely and precisely specified. Do not take this step lightly. If you do not know exactly what you want as the output of your program, you may be surprised at what your program produces. Be certain that you know what the input to the program will be, and exactly what information should be in the output, as well as what form that information should be in. For example, if the program is an accounting program for a bank, you must know not only the interest rate but also whether it is to be compounded annually, monthly, daily, or whatever. If the program is supposed to write poetry, you need to determine whether the poems can be in free verse or must be in iambic pentameter or some other meter.

*problem solving  
phase*

Many novice programmers do not understand the need to design an algorithm before writing a program in Pascal and so try to abbreviate the process by omitting the problem solving phase entirely or else reducing it to just the problem definition part. This seems reasonable. Why not “go for the mark” and save time? The answer is that *it does not save time!* Experience has shown that the two-phase process will produce a correctly working program faster. The two-phase process simplifies the algorithm-design phase by isolating it from the detailed rules of a programming language such as Pascal. The result is that the algorithm-design process becomes much less intricate and

---





**Figure 1.6**  
**Idealized program-**  
**design process.**

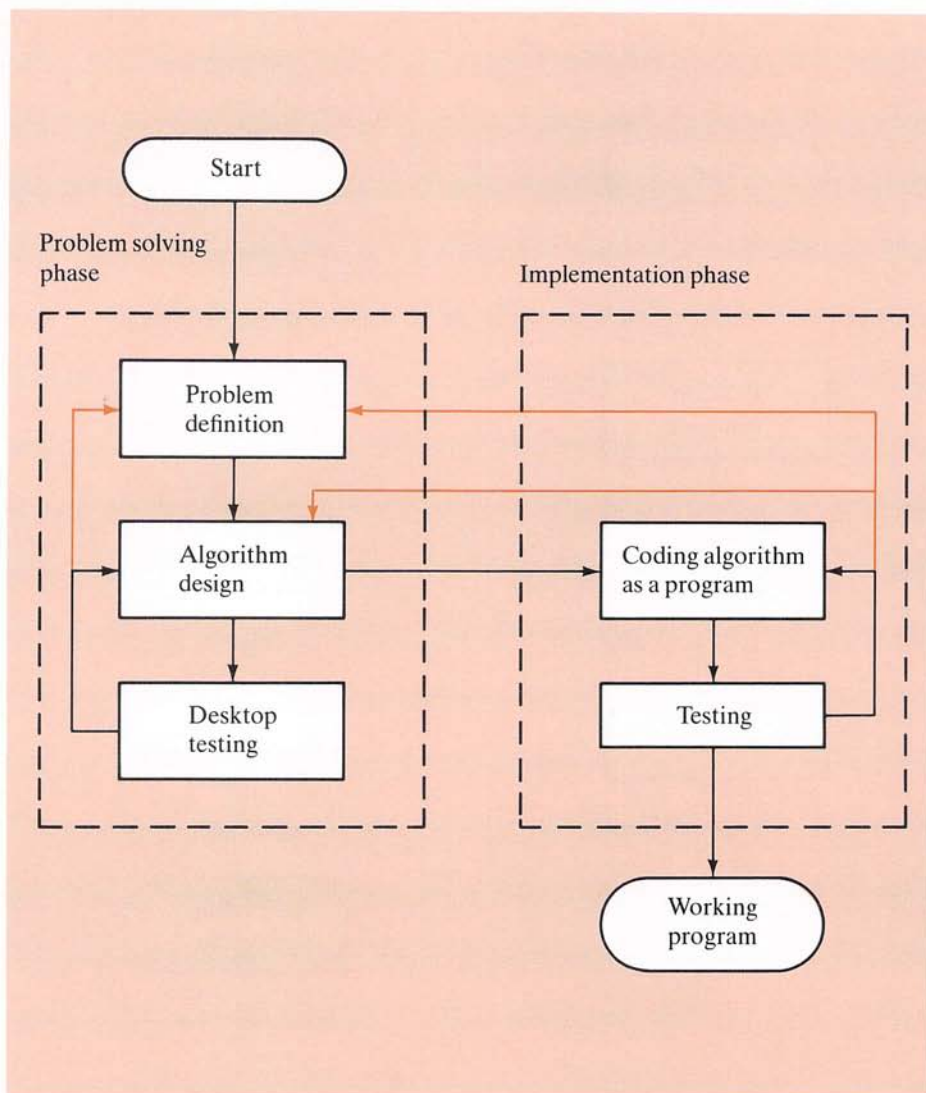
much less prone to error. For a modest-sized program it can represent the difference between a half day of careful work and several frustrating days of looking for mistakes in a poorly understood program.

The implementation phase is not trivial. There are details to worry about, details that occasionally can be quite subtle, but it is much simpler than you might at first think. Once you become familiar with Pascal, or any other programming language, the translation of an algorithm from English into the programming language becomes a routine task.

As indicated in Figure 1.6, testing takes place in both phases. The algorithm is tested and, if it is found to be deficient, it is redesigned. This first phase of testing is

*implementation  
phase*

*testing*



**Figure 1.7**  
Realistic program-  
design process.

performed by mentally going through the algorithm and executing the steps yourself. With large algorithms this will require the aid of pencil and paper. The Pascal program is tested by compiling it and running it on some sample data. The compiler will give error messages for certain kinds of errors. To find other types of errors, you must somehow check to see whether the output is correct.

The process depicted in Figure 1.6 is an idealized picture of the process. It is the basic picture you should have in mind, but reality is sometimes more complicated. In reality, mistakes and deficiencies are discovered at unexpected times, and you may have to back up, as shown in Figure 1.7. For example, testing the algorithm may reveal that



the problem definition was incomplete. In those cases, you must back up and reformulate the definition. Occasionally, deficiencies in the definition or algorithm may not be observed until a program is tested. If this occurs, you must back up and modify the definition or algorithm and all that follows it in the design process.

The next chapter contains an introduction to the Pascal language and some complete examples of this design process.

---

“And if you take one from three hundred and sixty-five,  
what remains?”

“Three hundred and sixty-four, of course.”

Humpty Dumpty looked doubtful. “I’d rather see that  
done on paper,” he said.

---

---

## Summary of Terms

**algorithm** Detailed, unambiguous, step-by-step instructions for carrying out a task.

**assembly language** A low level language that is almost the same thing as machine language. The only difference is that assembly language instructions are expressed in a slightly more readable form, instead of being coded as strings of zeros and ones. See *machine language*.

**auxiliary storage** Another name for *secondary memory*.

**code** Sometimes used to mean a program or part of a program.

**compiler** A program that translates programs from a high level language to machine language.

**CPU** The *central processing unit* of a computer. It performs the actual calculations and the manipulation of memory according to the instructions in a machine language program.

**data** This word has two meanings: (1) the input to an algorithm or program; (2) any information that is available to an algorithm or to a computer.

**editor** A program that allows the computer to be used as a typewriter. An editor also has a number of commands that are more powerful than those of a typewriter, such as ones that move an entire piece of text from one place to another.

**execute** When an instruction is carried out by a computer, either directly or in some translated form, the computer is said to execute the instruction. When a computer follows the instructions in a complete program, it is said to execute the program.

**file manager** Also sometimes called a *filer* or *file system*. A program that allows the user to store and retrieve objects called files. Among other things, a file can contain a Pascal program. Hence, a file manager is the program used to store and retrieve Pascal programs.

**hardware** The physical parts of a computer or computer system.

---

**high level language** A programming language that includes instructions that are more powerful than those found in machine language and that typically uses a grammar somewhat like English. Programs in a high level language usually cannot be executed directly by computers. See *machine language*.

**machine language** A language that can be executed directly by a computer. Programs in machine language consist of very simple instructions, such as to add two numbers. These simple instructions are coded as strings of zeros and ones. See *assembly language*.

**main memory** The memory that the computer uses as temporary “scratch paper” when carrying out a computation. See *secondary memory*.

**object program** The translated version of a program produced by a compiler. See *source program*.

**operating system** The program that controls and manages all other programs. You communicate with the computer through the operating system.

**program** An algorithm that a computer can either follow directly or translate and then follow the translated version.

**running a program** The process of giving a program and some data to a computer in such a way that the computer is instructed to carry out the program using the data.

**secondary memory** The memory a computer uses to store information in a permanent or semipermanent state. (When the computer does not have sufficient main memory for a computation, then secondary memory is also used as an addition to main memory.) See *main memory*.

**software** Another term for programs.

**source program** The input program to be translated by a compiler. See *object program*.

---

## Exercises

This book contains three kinds of exercises: Self-Test Exercises, Interactive Exercises, and Programming Exercises. The Self-Test Exercises are designed to provide you with a quick test of your understanding. The answers are relatively short and can be checked against the answers provided in the back of the book. The Interactive Exercises are designed to be done at the terminal. They are typically short programming exercises and are designed to give you a hands-on feel for the material. They may be done quickly; you need not worry too much about style details when working on them. The Programming Exercises are exercises that are suitable to be assigned as homework in a course. In this chapter, they ask you to produce algorithms in English. In subsequent chapters, they ask you to go through the entire design process and produce working Pascal programs.

### Self-Test Exercises

1. Write an algorithm to add two whole numbers. The input to the algorithm is to be two strings of digits representing the two numbers. For example, the number 1066 is
-

thought of as the four symbols 1-0-6-6. The algorithm should be capable of being followed by a child who has not yet learned to do addition.

2. Write an algorithm to tell whether an input word is a palindrome. A *palindrome* is a word that is the same spelled backwards and forwards, such as “radar.”
3. Write an algorithm to count the number of occurrences of each letter in an input word. For example, the input word “pop” contains two p’s and one o.

### Interactive Exercise

4. A good illustration of an algorithm is the instruction set for the U.S. Internal Revenue Service’s long form 1040. If you have a copy readily available, read it through, noticing how very explicit the instructions are.

### Programming Exercises

5. Write an algorithm to multiply two whole numbers. The rules are the same as in Exercise 1.
6. Write an algorithm to subtract one whole number from another. The rules are the same as in Exercise 1. (This is harder than it sounds.)
7. Write an algorithm to divide one whole number by another whole number. The rules are the same as in Exercise 1.
8. Write an algorithm that takes a page of text as data (input) and corrects the spacing according to the following rules: There should be exactly one space between two adjacent words, except that there are two spaces between adjacent sentences, and paragraphs are indented by exactly three spaces. Define the start of a paragraph as an indentation of one or more spaces at the start of a line. The data may contain any number of spaces, except that you may assume that there are no spaces inside of words and that there is at least one space between any two words on the same line. The output is to be written onto a second sheet of paper.
9. Many banks and savings and loan institutions compute interest on a daily basis. On a balance of \$1000 with an interest rate of 6%, the interest earned in one day is 0.06 multiplied by \$1000 and then divided by 365, because it is only for one day of a 365-day year. This yields \$0.16 in interest, and so the resulting balance is \$1000.16. The interest for the second day will be 0.06 multiplied by \$1000.16 and then divided by 365. Design an algorithm that takes three inputs: the amount of a deposit, the interest rate, and a duration in weeks. The algorithm then calculates the account balance at the end of the duration specified.
10. Negotiating a consumer loan is not always straightforward. One form of loan is the discount installment loan, which works as follows. Suppose a loan has a face value of \$1000, the interest rate is 15%, and the duration is 18 months. The interest is computed by multiplying the face value of \$1000 by 0.15 to yield \$150. That figure is then multiplied by the loan period of 1.5 years to yield \$225 as the total interest owed. That amount is immediately deducted from the face value, leaving the consumer with only \$775. Repayment is made in equal monthly installments based on the face value. So the monthly loan payment will be \$1000 divided by 18, or \$55.56. This method of calcula-



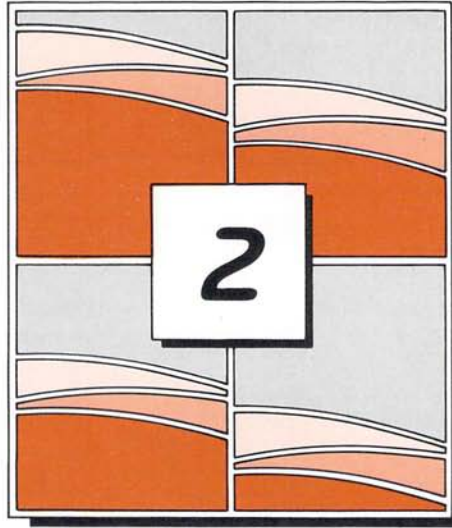
tion may not be too bad if the consumer needs \$775, but the calculation is a bit more complicated if the consumer needs \$1000. Design an algorithm that takes three inputs: the amount the consumer needs to receive, the interest rate, and the duration of the loan in months. The algorithm should then calculate the face value required in order for the consumer to receive the amount needed and should also calculate the monthly payment.

---

## References for Further Reading

- H.L. Capron, *Computers: Tools For an Information Age*, 1987, Benjamin/Cummings, Menlo Park, Ca. A simple introduction to programming systems and modern uses of computers. This book would be good if you are completely bewildered by Chapter 1.
- L. Goldschlager and A. Lister, *Computer Science—A Modern Introduction*, 1982, Prentice-Hall International Series in Computer Science, Prentice-Hall, Englewood Cliffs, N.J. Discusses many of the issues in this chapter in greater detail. It is written at the introductory level.
- T. Kidder, *The Soul of a New Machine*, 1981, Avon Books, New York. This is a popular description of the engineering effort that went into designing a specific computer. It is entertaining and you do pick up a few technical facts as well.
- R.E. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming*, 1981, John Wiley, New York. An introduction to algorithms and programming in a very simple setting.
- I. Pohl and A. Shaw, *The Nature of Computation, An Introduction to Computer Science*, 1981, Computer Science Press, Rockville, Md. Discusses many of the issues in this chapter in greater detail. It is written at the introductory level.
-





## ***Introduction to Problem Solving with Pascal***

Once a person has understood the way variables are used in programming,  
he has understood the quintessence of programming.

*E.W. Dijkstra, Notes on Structured Programming*

## Chapter Contents

The Notion of a Program Variable  
Stepping Through a Program  
Assignment Statements  
Pitfall—Uninitialized Variables  
Data Types—An Introduction  
More about Real Values  
Type Compatibility  
Arithmetic Expressions  
Simple Output  
Input  
Designing Input and Output  
Pitfall—Input in Wrong Order  
Names: Identifiers  
Putting the Pieces Together  
TURBO Pascal  
Introduction to Programming Style  
Blaise Pascal (Optional)

Self-Test Exercises  
Interactive Exercises  
Problem Solving and Program Design  
Top-Down Design  
Case Study—A Guessing Game  
Integer Division—mod and div  
Desktop Testing  
Case Study—Making Change  
Exploring the Solution Space  
Summary of Problem Solving Techniques  
Summary of Programming Techniques  
Summary of Pascal Constructs  
Exercises

**I**n this chapter we introduce the Pascal programming language. We explain some sample programs and present enough details of the Pascal language to allow you to write simple programs of your own. We continue the discussion of problem solving that we began in Chapter 1 by presenting some fundamental techniques for designing algorithms. Finally, we illustrate these design techniques by developing two Pascal programs from the problem-definition stage through to a final working program.

---

## The Notion of a Program Variable

Programs manipulate data, such as numbers and letters. In order to store the data produced by subcomputations, and in order to have a way to name the data, Pascal and many other common programming languages use objects called *variables*. Variables are the very heart of a programming language like Pascal. These programming variables have some similarity to the variables used in algebra and related branches of mathematics, but they very definitely are not the same. To start with, programming variables traditionally have longer names than the variables used in algebra. In a Pascal program, a variable is written as a string of letters and digits that begins with a letter. Some sample names for Pascal variables are

```
X Y Z SUM N1 N2 SALLY Joe rate Time
```

Pascal variables are things that can hold numbers or other types of data. For the moment, we will confine our attention to variables that hold only numbers. These variables are rather like small blackboards on which a number can be written. Just as the number written on a blackboard can be changed, so too can the number held by a Pascal variable be changed. Just like blackboards, variables might contain no number at all, or they might contain the number left there by the last person or thing that used the variable. The words *hold* and *contain* when applied to variables are exact synonyms and, if you think in terms of the blackboard analogy, refer to the item written on the figurative blackboard. The number or other type of data held in a variable is called its *value*.

*variables  
and  
values*

Most compilers translate variables into memory locations. A memory location is assigned to each variable, and the value of the variable, in a coded form consisting of zeros and ones, is kept in that location. For example, Figure 2.1 contains a Pascal program that uses the three variables N1, N2, and SUM. When that program is compiled, the compiler might assign the locations 1001, 1002, and 1003 to the variables N1, N2, and SUM, respectively. When the value of a variable is changed, the coded number in its assigned memory location changes to the new number. Whether or not your particular compiler actually makes this assignment of memory locations to variables is irrelevant, since it will make the program act as if just such an assignment had been made.

*memory  
locations  
and  
variables*

To illustrate how program variables can change their value, we will give a step-by-step explanation of the program displayed in Figure 2.1. Unless otherwise noted, we will always assume that input is via a keyboard and that the output is via a display screen.

---

## Stepping Through a Program

In Figure 2.1, the eight lines between *begin* and *end* contain eight instructions to be carried out by the computer. Such instructions are called *statements*. The semicolons at the ends of the statements are used to separate the statements and, strictly speaking, are not part of the statements themselves. That is why the last statement is not followed

*statements*

### Program

```
program Sample(input, output);  
var N1, N2, SUM: integer;  
begin  
  writeln('Enter two numbers');  
  readln(N1, N2);  
  SUM := N1 + N2;  
  writeln(N1, ' Plus ', N2, ' Equals ', SUM);  
  writeln('Enter another two numbers');  
  readln(N1, N2);  
  SUM := N1 + N2;  
  writeln(N1, ' Plus ', N2, ' Equals ', SUM)  
end.
```

### Sample Dialogue

```
Enter two numbers  
4 5  
4 Plus 5 Equals 9  
Enter another two numbers  
8 9  
8 Plus 9 Equals 17
```

**Figure 2.1**  
**A Pascal program.**

by a semicolon; there is no subsequent statement from which it needs to be separated. Minor points of punctuation like this need not be a serious concern for us yet. For now, simply note that the semicolons must be there. If you prefer, it is perfectly acceptable to add a semicolon after the last statement and thereby make things uniform. The program will perform identically with or without the extra semicolon.

The word `writeln` in the first line is pronounced “write-line.” This first statement does not affect any variables. It simply causes the following phrase to appear on the screen:

```
Enter two numbers
```

Suppose that in response to this an obedient user types two numbers on the keyboard, say `4`, followed by a space, followed by a `5`, followed by pressing the return key. (The return key is the one that starts a new line, much like the carriage return on a typewriter.)

The next statement, shown below, tells the computer what to do with these numbers.

```
readln(N1, N2)
```

(We will usually not bother to show the final semicolon when displaying lines out of a program like this.) The word `readln` is pronounced “read-line.” This statement instructs the computer to “read” the two numbers into the variables `N1` and `N2`. It causes the value of the variable `N1` to become equal to the first number that was typed, namely `4`, and the value of `N2` to become equal to the second number, namely `5`. Recall that

---



our hypothetical compiled translation of the Pascal program assigned memory location 1001 to N1 and 1002 to the N2. This means that location 1001 in the computer's memory will contain a coded version of the number 4. Similarly, location 1002 will contain the code for 5. Prior to this time, the variables N1 and N2 had no value.

The next statement in our sample program is

```
SUM := N1 + N2
```

The meaning of the part `N1 + N2` is what you might guess. It instructs the computer to add the value of N1 to the value of N2. In other words, add 4 to 5. When the translated version of this statement is executed, the computer (the CPU, to be precise) will retrieve the 4 from location 1001 and the 5 from location 1002 and add them together to obtain 9. This statement also specifies what is to become of the resultant number 9. It becomes the value of the variable SUM and so goes into the location assigned to SUM, namely 1003.

The pair of symbols `:=` is called the *assignment operator*, and statements such as the one under discussion are called *assignment statements*. The assignment operator is composed of two symbols, a colon and an equal sign, but it is considered to be one item and is always treated as a single unit. In particular, there should be no space between the colon and the equal sign. The assignment operator has no standard pronunciation. Many people use the phrase “gets the value,” which describes the action; others simply pronounce it “assignment operator.” Some people even read it literally as “colon equals,” but the meaning of the assignment operator is not derived from the usual meaning of the colon and equal symbols. The `:=` instructs the computer to change the value of the variable on the left-hand side to the value given on the right-hand side. We will say more about the assignment operator shortly, but first let us explain the rest of our sample program.

*assignment  
operator*

The next statement is

```
writeln(N1, ' Plus ', N2, ' Equals ', SUM)
```

It shows the result of the assignment statement by causing the following phrase to appear on the screen:

```
4 Plus 5 Equals 9
```

For the moment, do not worry about why the word `writeln` is embellished with a funny ending or why words like `' Plus '` are surrounded with blanks and single quotes. We will come back to those details shortly. For now, simply note that the values of the variables and the words in quotes are output to the screen in the order in which they occur in the `writeln` statement.

The rest of the program is almost a repetition of what we have just discussed. The program requests two more numbers with the statement

```
writeln('Enter another two numbers')
```

This time, let us suppose that the user types in 8 and 9, again separated by a blank, and terminates the line by pressing the return key. The two numbers are read when the program executes the next statement in the program:

```
readln(N1, N2)
```

*changing  
the value  
of a  
variable*

This second `readln` statement sets the value of `N1` equal to 8 and the value of `N2` equal to 9. The old values of 4 and 5 are lost.

Next comes the second assignment statement:

```
SUM := N1 + N2
```

This changes the value of `SUM` to the value of `N1` plus the value of `N2`. In other words, the value of `SUM` is changed to 8 plus 9, or 17. The old value of `SUM` is lost. The final `writeln` statement outputs the new values of `N1`, `N2`, and `SUM`.

Figure 2.1 also shows the complete dialogue between the computer and the user as it would appear on the screen. The material typed by the user and the output produced by the program are shown in different typefaces, which helps to clarify where the different lines came from. In reality, they would appear in the same typeface.

---

## Assignment Statements

### *expression*

An *assignment statement* always consists of a variable on the left-hand side of the assignment operator and an expression on the right-hand side. The expression may be a variable, a number, or a more complicated expression made up of variables, numbers, and arithmetic operators such as the plus sign and the minus sign. This statement instructs the computer to evaluate the expression on the right-hand side and to set the value of the variable equal to that value of the expression. A few more examples may help to clarify the way these statements work.

In Pascal the traditional multiplication sign is not used and multiplication is represented by an asterisk, as shown in the following assignment statement:

```
SUM := N1 * N2
```

This statement is just like the assignment statements in our sample program except that it performs multiplication rather than addition. The statement changes the value of `SUM` to the product of the values of `N1` and `N2`. Of course, that makes `SUM` a poor choice for the name of our variable, but the program will still run. When the computer sees the identifier `SUM`, it knows it is a variable capable of storing numbers, but it does not know or care that it spells an English word indicating addition.

The expression on the right-hand side of an assignment statement can simply be another variable. The statement

```
N1 := N2
```

changes the value of the variable `N1` to that of the variable `N2`. The value of `N2` is not affected. This example is illustrated in Figure 2.2.

As yet another example, the following changes the value of `N2` to 3:

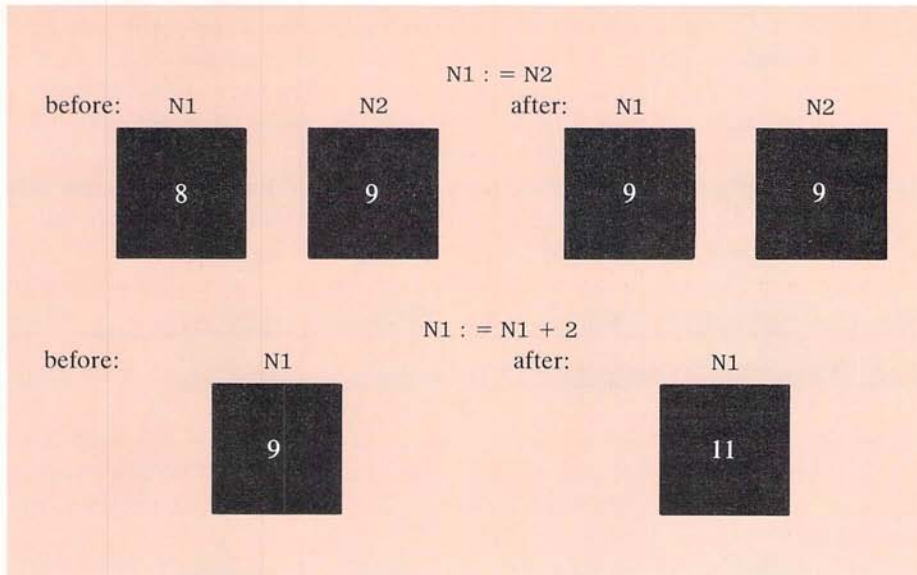
```
N2 := 3
```

### *constants*

The number 3 on the right-hand side of the assignment operator is called a *constant* because, unlike a variable, its value cannot change.

Because variables can change value over time, and because the assignment operator is one vehicle for changing their values, an element of time is involved in the

---



**Figure 2.2**  
The assignment operator.

meaning of an assignment statement. First, the expression on the right-hand side of the assignment operator is evaluated. After that, the value of the variable on the left-hand side of the operator is changed to the value obtained from that expression. This means that a variable can meaningfully occur on both sides of an assignment operator. As an example, consider:

$N1 := N1 + 2$

This statement will increase the value of N1 by 2.

## Pitfall

### Uninitialized Variables

*A variable on the right-hand side of an assignment operator must have been given a value before the assignment statement is executed.* Until a program gives a value to a variable, that variable has no value. For example, if the variable X has not been given a value, either in an assignment statement or in a read or readln statement (and if it has not been given a value by any of the methods still to be discussed), then the following is an error:

$Y := X + 1$

This is because X has no value and so the expression  $X + 1$  has no value. A variable that has not been given a value is said to be *uninitialized*.

What happens in this situation depends on your particular system. You may get an error message warning you that X has been used without being initialized. In other, less ideal situations, the value of X may simply be set to some random



quantity determined by whatever was left in the computer's memory by the previously run program. Hence, it is possible that when the program is run twice with the same input data it may give two different outputs. Whenever a program gives different output on the same data and without any change in the program itself, you should suspect an uninitialized variable.

Some other programming languages automatically initialize all numeric variables to zero. Although this may be true of certain versions of Pascal, you cannot and should not count on it.

## Data Types—An Introduction

A *data type* is what the words say it is, namely a type or category of data. Each variable can hold only one type of data. In our sample Pascal program all variables were of type *integer*, which means that their values must be integers. An integer is any whole number, such as

38 0 1 89 3987 -12 -5

The value of a variable of type *integer* cannot be a fraction. The variables N1, N2, and SUM in the program in Figure 2.1 can never contain fractions like 1/2 or 3.1416. Fractions belong to another data type called *real*, which we will discuss later in this section.

*variable  
declarations*

Every variable in a Pascal program must be *declared*, that is, the type of the variable must be stated. This is done at the beginning of the program. In our first sample program the variables were declared by the line

`var N1, N2, SUM: integer;`

The declaration consists of the word *var*, followed by one or more blanks, followed by a list of variables separated by commas, followed by a colon, followed by a type name, and finally ended with a semicolon. Extra blanks may be added as long as you do not insert blanks in the middle of words such as *var* and *integer* or in the middle of variable names.

There are two reasons for requiring these declarations: to clarify your thinking by reminding you of the type of data the variable will hold and to provide information to the compiler. Recall that the computer has only strings of zeros and ones in memory. In order to treat these strings as integers, it uses a code to encode each integer as a string of zeros and ones. It uses a different code to encode letters as strings of zeros and ones. The declaration tells the compiler, and ultimately the computer, which code to use.

*reals*

Numbers that include a fractional part, such as the ones below, are of type *real*:

2.71828 0.098 -15.8 100053.98

*number  
constants*

When such numbers appear in a Pascal program they are called *constants*. All numeric constants have a type, either *integer* or *real*. Conceptually, every whole number is both an integer and a real number. However, the computer makes a distinc-



tion between whole numbers of type `integer` and whole numbers of type `real`. In particular, the constant for the `integer` three is written `3`, whereas the constant for the `real` number three is written `3.0`. We will have more to say about this distinction in the next section.

Just as for numeric constants, every numeric variable is either of type `integer` or of type `real`. A variable `Z` is declared to be of type `real` in the following way:

```
var Z: real;
```

Division is one way to obtain values of type `real`. In Pascal division can be expressed using a slash. So `N1/N2` means to divide the value of `N1` by the value of `N2`. The result of this operation is always of type `real` no matter what the values or types of the two numbers are. Hence, in the following assignment statement, the variables `N1` and `N2` might be of type `integer` or of type `real`, but the variable `Z` can only be of type `real`:

```
Z := N1/N2
```

The type for letters or, more generally, any single symbol is `char`, which is short for “character.” Values of this type are frequently called *characters* in books and in conversation, but in Pascal programs this type must always be spelled in the abbreviated fashion `char`. Two variables `X` and `Y` of type `char` are declared as follows:

```
var X, Y: char;
```

A variable of type `char` can hold any character on the input keyboard. They may each hold only one character; they cannot hold strings containing more than one character. So, for example, `X` could hold an `'A'` or a `'+'` or an `'a'`. If both upper- and lower-case letters are available, they are considered to be different characters.

The single quotes indicate that we literally mean the letter. Hence, `X` is used for a variable named `X`, whereas `'X'` is used for the uppercase version of the third from the last letter of the alphabet. This is an important distinction. For example, the statement

```
Y := X
```

changes the value of the variable `Y` to the value of the variable `X`. If `X` contained the letter `'A'`, then this statement will change the value of `Y` to `'A'`. On the other hand, the statement

```
Y := 'X'
```

changes the value of the variable `Y` to the letter `'X'`, which is quite another thing. The program in Figure 2.3 illustrates this important distinction.

Unlike some printed texts, Pascal has only one kind of single quote; the opening quote and the closing quote are the same symbol.

Expressions consisting of a character in single quotes, such as `'X'`, are also called *constants* and are in fact the same sort of objects as the numeric constants, such as `3` and `5.98`, except that they are of a different type, specifically the type `char`. Be sure to note that while `'3'` and `3` are both constants, they are very different constants. The first is a character, a mere symbol. It is of type `char` and may be used on

*characters*  
(*char*)

*quotes*

**Program**

```

program Tricky(input, output);
var X, Y: char;
begin
  X := 'A';
  Y := X;
  writeln('The first value of Y is:');
  writeln(Y);

  Y := 'X';
  writeln('The second value of Y is:');
  writeln(Y);

  writeln('I hope this helped to explain quotes.')
end.

```

**Sample Dialogue**

```

The first value of Y is:
A
The second value of Y is:
X
I hope this helped to explain quotes.

```

**Figure 2.3**  
Using single quotes  
for characters.

the right-hand side of an assignment statement to give a value to a variable of type `char`. The constant `3` is a number. It is of type `integer` and may be used to give a value to a variable of type `integer`. There are constants of other types as well, and we will discuss them as the opportunities arise.

*sample  
declarations*

It is perfectly acceptable to have variables of more than one type in a program. In such cases they are all declared at once following the format illustrated below:

```

var N1, N2: integer;
    Time: real;
    Initial: char;

```

Note that the word `var` is only used once, no matter how many variables are declared and no matter how many variable types are used in the declarations.

The variables need not be declared in any particular order. For instance, the above declaration is equivalent to the following one:

```

var Initial: char;
    N1: integer;
    Time: real;
    N2: integer;

```

The best order is the one that groups the variables according to their use in the program.

---

## More about Real Values

Conceptually, a whole number is a special kind of real number, namely one that happens to have only zeros after the decimal point. If that were the only difference between the two types, we would never need to use the type `integer`. There is another important difference, however. Numbers of type `integer` are stored as exact values, while numbers of type `real` are stored only as approximate values. Thus, if we know that a certain variable will always contain a whole number, then it is best to make it of type `integer`. The precision with which real values are stored varies from one computer to another, but on most systems the extra digits in the following assignment statement are pointless:

```
X := 3.14159265358979323846
```

The program is likely to give exactly the same output if the `real` value is changed to

```
X := 3.14159
```

Numeric constants of type `real` are written differently from those of type `integer`. Constants of type `integer` must not contain a decimal point. Constants of type `real` may be written in two different forms. The simple form for real constants is like the everyday way of writing decimal fractions. When written in this form, a real constant must contain a decimal point and there must be at least one digit before and one digit after the decimal point. No number in Pascal may contain a comma. Hence, none of the following are allowed as constants of type `real` (nor as constants of type `integer`):

```
1,000 .009 -.05 72.
```

In addition to the simple notation that we have been using for real constants, there is another, more complicated notation for expressing constants of type `real`. This notation is frequently called *scientific notation* or *floating point notation*. It is particularly handy for writing very large numbers and very small fractions. For instance, the numbers

$$3.67 \times 10^{17} = 367000000000000000.0$$

and

$$5.89 \times 10^{-6} = 0.00000589$$

are best expressed in Pascal by the constants `3.67E17` and `5.89E-6`, respectively. The E stands for *exponent* and means “multiply by 10 to the power that follows.” This *E notation* is used because keyboards normally do not have any way to enter exponents as superscripts.

Another way of understanding E notation is to think of the number after the E as telling you to move the decimal point to the right that many places. For example, to change `3.49E4` to a numeral without an E, you move the decimal point 4 places to the right and obtain `34900.0`. If the number after the E is negative, then move the decimal point the indicated number of spaces to the left, inserting extra zeros if need

*reals  
versus  
integers*



*E  
notation  
syntax*

be. For example,  $3.49\text{E}-2$  and  $0.0349$  are two ways of writing the same number.

There are rigid rules for writing constants in the E notation. The number before the E can be any decimal number, with or without a plus or minus sign. It need not contain a decimal point, but if it does, there must be at least one digit before and at least one digit after the decimal point. The number after the E is called the *exponent* and must be a whole number, either positive, negative, or zero. It cannot contain a decimal point. Hence, the following are all correctly formed constants of type *real*:

```
9.34E13  5E27  5E-27  -8.62713E21
1.234E-15  -6.783E-12  1.0E+13  +34.78E56
```

By contrast, none of the following are acceptable constants:

```
.5E12  -.7E13  3.5E22.5
```

The first two are incorrect because they have no digit before the decimal point. The last one is incorrect because it has a decimal point in the exponent.

---

## Type Compatibility

*integers  
in place of  
reals*

In Pascal programs,  $2$  and  $2.0$  are different kinds of numbers: the first is of type *integer* and is an exact value; the second is of type *real* and represents an approximate value. This is not always an important issue, since the computer will perform an automatic type conversion, converting a number of type *integer* to an approximately equal number of type *real* whenever the situation demands a value of type *real*. Hence, if *X* is a variable of type *real*, then the following is allowed:

```
X := 2
```

The reverse situation is not allowed, however. If *Y* is of type *integer*, then the following is illegal in Pascal:

```
Y := 2.0
```

This sort of type conflict is not likely to arise in such a simplistic manner, but the same principle applies in more subtle situations. To take a slightly more likely mistake, note that the following is illegal whenever *X* is of type *real* and *Y* is of type *integer*:

```
Y := X
```

---

## Arithmetic Expressions

Constants and variables of the types *integer* and *real* may be combined to form more complex expressions by using the operators  $+$ ,  $-$ ,  $*$ , and  $/$  for addition, subtraction, multiplication, and division, respectively. We have already used simple versions of such arithmetic expressions. By using parentheses, it is possible to build more complicated expressions from these simpler expressions.

---



As an example, suppose that  $N$ ,  $M$ , and  $Y$  are variables and that each one is either of type `real` or type `integer`. The right-hand side of the following assignment statement is then a well-formed arithmetic expression:

$$X := 2 * (N + (M/3) + 4 * Y)$$

The value of the expression is `real` because it contains a division and the result of a division is always of type `real`. Since the expression is of type `real`, the variable  $X$  on the left-hand side of the assignment operator must also be of type `real`.

Any combination of the types `real` and `integer` may be used with the operators  $+$ ,  $-$ ,  $*$ , and  $/$ . The type of the values resulting from the arithmetic expression is determined according to the following simple rule: If the operators can ever be used on *any* values of the types being combined so as to produce a number with a nonzero value after the decimal point, then the result is of type `real`; otherwise, it is `integer`. A more detailed statement of the rule is given in Figure 2.4, but this short rule is easier to remember.

*mixing  
reals and  
integers*

Notice that the type of an arithmetic expression is determined by the types of its subexpressions and by the operations it contains, not by the particular value of the expression. Whether an arithmetic expression evaluates to a “whole number” or not is irrelevant to the type of the expression. The quantity  $4/2$  is of type `real` despite the fact that the answer “comes out even.”

Any reasonable spacing will do in arithmetic expressions. You can insert spaces before and after operations and parentheses, or you can omit them. Do whatever produces a result that is easy to read.

The order of operations can always be determined by parentheses, as illustrated in the following two expressions:

*parentheses*

$$\begin{aligned} (X + Y) * Z \\ X + (Y * Z) \end{aligned}$$

To evaluate the first expression, add  $X$  and  $Y$  and then multiply the result by  $Z$ . To evaluate the second expression, multiply  $Y$  and  $Z$  and then add the result to  $X$ .

### How to Determine the Type of an Arithmetic Expression

1. Combining something of type `real` with something either of type `integer` or type `real` always yields something of type `real`.
2. Combining anything with the division sign,  $/$ , always yields something of type `real`.
3. Combining two things of type `integer` with either the addition sign,  $+$ , the subtraction sign,  $-$ , or the multiplication sign,  $*$ , yields something of type `integer`.
4. Placing a minus sign,  $-$ , in front of an arithmetic expression does not change its type.

**Figure 2.4**  
Type rules for  
arithmetic  
expressions.

*precedence  
rules*

If you omit parentheses, the computer will follow precedence rules similar to those used in everyday arithmetic. For example,

$$X + Y * Z$$

is evaluated by first performing the multiplication and then the addition. Except for some very standard cases, such as a string of additions or a simple multiplication embedded inside an addition, it is best to use parentheses, even if the intended order of operations is the one dictated by the precedence rules. The parentheses make the expression easier to read and less prone to programmer error. The exact precedence rules for the operations we have seen so far are given in Figure 2.5. A complete set of Pascal precedence rules are given in Appendix 3.

### Order of Evaluation for Arithmetic Expressions

1. If parentheses are present, they determine the order of operations.
2. If the order is not determined by parentheses, then multiplication and division operations are evaluated before addition and subtraction operations.
3. If there is still a question as to which operation to perform first, the competing operations are performed from left to right.

**Figure 2.5**  
Precedence rules  
for arithmetic  
expressions.

Unlike written and printed mathematical formulas, which may contain square brackets and various other forms of parentheses, Pascal allows only one kind of parentheses in arithmetic expressions. The other varieties are reserved for other purposes.

*exponents*

While we are on the subject of arithmetic operations, we should point out that Pascal contains no operator for exponentiation. There is no Pascal equivalent of  $x^y$ .

Mathematical Formula	Pascal Expression
$b^2 - 4ac$	$B * B - 4 * A * C$
$x(y + z)$	$X * (Y + Z)$
$\frac{1}{x^2 + x + 1}$	$1 / (X * X + X + 1)$
$\frac{a + b}{c - d}$	$(A + B) / (C - D)$

**Figure 2.6**  
Pascal arithmetic  
expressions.

Later on we will see how to define expressions that are equivalent to this kind of exponentiation. For now, you will simply have to use repeated products to obtain powers and do without any equivalent of fractional exponents. Thus, X cubed is expressed as

`X*X*X`

Figure 2.6 shows some examples of common arithmetic expressions and how they would be expressed in Pascal.

---

## Simple Output

The values of variables as well as strings of text may be output to the screen with `writeln` statements, such as those used in our sample program in Figure 2.1. Any combination of variables of different types as well as strings may be output. The strings of characters are enclosed in single quotes. The items to be output are listed in the order in which they are to appear on the screen.

The computer will not insert extra space before or after strings or the values of variables of type `char`, which is why the quoted strings in the samples usually start and end with a blank. The blanks keep the various strings and numbers from running together.

Values of type `real` are often output in E notation. For example, if X is a variable of type `real`, then the statements

```
X := 1234.56;
writeln(X)
```

are likely to produce output that looks something like the following:

```
1.23456000000000E03
```

which means

```
1.23456000000000 × 103
```

and is another way of expressing the number 1234.56.

There are some rules about quoted strings that you must observe when you include them in `writeln` statements, or anywhere else for that matter. Remember that single quotes are used and that both the opening and closing quote are the same symbol. If you want to include a single quote symbol within a quoted string, then you must use two single quotes; otherwise the computer will interpret the single quote as marking the end of a quoted string. For example, the output of

```
writeln('Surf 's Up')
```

is the following:

```
Surf 's Up
```

The suffix `ln` on the word `write` is short for “line.” It is usually pronounced “line” but it is always spelled `ln`. Including the `ln` instructs the computer to start a new line *after* writing out the listed items. For example, consider the two statements

*writeln*

*blanks*

*outputting  
reals*

*quotes  
inside of  
quotes*

*write  
versus  
writeln*



```
writeln('First line');  
writeln('Second line')
```

They cause the following to appear on the screen:

```
First line  
Second line
```

On the other hand, consider the two statements

```
write('First line');  
writeln('Second line')
```

They cause the following to appear on the screen:

```
First lineSecond line
```

The last output statement in a program should always be a `writeln` rather than a `write`. This is because some systems require that the end of all lines, including the last line of output, be explicitly indicated.

While we are on the subject, we should note two minor but useful properties of `writeln`. First, you can use `writeln` without specifying anything to output. This simply causes the computer to skip to the next line, which is handy for skipping lines and for when you do use `write`. Hence, the three statements

```
write('Line one');  
writeln;  
writeln('Line two')
```

are equivalent to the two statements

```
writeln('Line one');  
writeln('Line two')
```

The second property, one that is minor but occasionally useful, is that you can place expressions within a `write` or `writeln` statement. Thus, the following is permitted; its meaning is obvious:

```
writeln(N1, ' Plus ', N2, ' Equals ', N1 + N2)
```

---

## Input

### *readln*

`read` and `readln` are used for input and are analogous to `write` and `writeln`. They are written the same way, namely with a list of items separated by commas and enclosed in parentheses. In this case, all the items must be variables. They instruct the computer to read input values and to set the values of the variables equal to the values read in. We will assume that the input is entered from a keyboard, but the details are similar for input obtained from other sources.

Variables of the types `integer`, `real`, and `char` may be given values with a `read` or `readln` statement. Later we will introduce other data types besides these

---



three. On most systems, however, only values of type `integer`, `real`, and `char` may be read in from the keyboard.

You may mix the types of the variables in a single `read` or `readln` statement, but the types of the values input should match the types of the variables listed in parentheses. If a variable is of type `integer`, then the number entered for that variable should be an integer constant such as 12 or -7; it should not be a real such as 12.5. Even using 12.0 can produce serious problems. If a variable is of type `real`, then the number typed in for that variable may be of type either `real` or `integer`. The computer will automatically convert an `integer` value to an approximately equal `real` value when filling variables of type `real`. Of course, you should never try to fill a variable of type `integer` or `real` with a character such as a letter of the alphabet.

When entering numbers, you must insert one or more blanks between numbers on the same line so that the computer knows where one number ends and the next begins. The situation with characters is different. Anything you type, *including a blank*, is a value of type `char`. Hence, there is no space before an input character that is intended for a variable of type `char`.

*blanks  
in  
input*

The difference between `read` and `readln` is that `readln` instructs the computer to go to the next line *after* reading values for all variables, thus causing the rest of the input on the current line to be discarded. The exact details are different for numbers and for character input.

*read  
versus  
readln*

When reading in numbers with either `read` or `readln`, the computer will automatically go to the next line when it needs more values than are available on the current line. The only difference between the two is that with `read` the computer does not begin reading the next line until the data on the current line is exhausted. On the other hand, after the computer completes a `readln`, it discards anything remaining on the current line, and any subsequent `read` or `readln` will take its first value from the next line.

When reading in characters, `readln` causes a similar discarding of the remainder of the current line; any subsequent `read` or `readln` will take its values from the next line of input. However, with character input, the computer does not automatically go to the next line when the current line is exhausted. For data of type `char`, there should normally be at least as many symbols on a line as the program expects. A `readln` should be used to instruct the computer to go to the next line when the current line is exhausted.

A simple `readln`, without any variables, instructs the computer to disregard the rest of the input on the current line. For example, the two statements

```
read(X, Y);  
readln
```

are equivalent to the single statement

```
readln(X, Y)
```

Uses for the unadorned `readln` will occur to you naturally as your programming skills develop.

When writing your first few programs, it is probably best to avoid `read` and `write` in favor of `readln` and `writeln`. Using `writeln` and `readln` to organize the screen display into alternating lines of input and output is usually easier and safer than trying to integrate `read` and `write` instructions.

---

## Designing Input and Output

*prompt  
lines*

When the computer executes a `read` or `readln`, it expects data to be entered at the keyboard. If none is typed in, it simply waits for it. The program must tell the user when to type in data; the computer will not automatically ask for the data. That is why the sample programs contain statements like the following:

```
writeln('Enter two numbers')
```

*echoing  
input*

These output statements *prompt* the user to enter input.

When the user is entering input from a terminal, the input appears on the screen as it is typed. Nonetheless, the program should always write out the input values at some point before the program ends. This is called *echoing* the input, and it serves as a check to see that the input was read in correctly. Just because the input looks good on the screen does not mean that it was read correctly. There could be an unnoticed typing mistake, or the input could be read in incorrectly because the line breaks are not where the program expects them to be. Echoing input serves as a test of the integrity of the input data.

---

## Pitfall

### Input in Wrong Order

A human being given two numbers that represent height and weight will figure out which number is which no matter what order the numbers are given in. Nobody is 180 feet tall and weighs 6 pounds. A computer, however, makes no such test for reasonableness and will happily accept such numbers for height and weight. Therefore, you must always be careful that your programs instruct the user to input values in the correct order, or the program is likely either to terminate abnormally or to produce incorrect results.

---

## Names: Identifiers

In Pascal, variables are written as strings of symbols. The string of symbols serves as the name of the variable. Pascal programs also have names. The name of the program in Figure 2.3 is *Tricky*. As we learn more Pascal, we will encounter other Pascal objects that have names.

---



A name used in a Pascal program is called an *identifier* and is defined to be any string of letters and digits, provided the string starts with a letter. With the exception of one special class of identifiers that we will discuss later in this section, any Pascal identifier can be used as the name of a variable in a Pascal program. The following are all examples of Pascal identifiers:

```
X X1 sample ABC123z7 SUM Data1 Data2 TEMP
```

The following are not Pascal identifiers:

```
12 3X DATA.1 FILE.TEXT DATA-GOOD
```

The first two are not identifiers because they start with a digit rather than a letter. The remaining three are not identifiers because they contain symbols other than letters and digits.

The Pascal standard places no limit on the length of a Pascal identifier. However, many compilers will ignore all characters after some specified number of initial characters. From now on, we will assume that the computer ignores all but the first eight characters. Even under this assumption, the following is a perfectly valid identifier:

```
TheFinalAnswer
```

In some contexts, this may be the most sensible identifier to use, but if you use such long identifiers, be certain that there is no other identifier with the same first eight letters. On some compilers, the following three identifiers are for all practical purposes equal, because their first eight letters are the same:

```
TheFinalAnswer TheFinalResult TheFinal
```

There is nothing special about the number eight, but there often is some limit. The first versions of Pascal used eight as the limit. Many implementations still use this limit, and no system uses a limit of less than eight characters. For these reasons, it is a good idea to pretend that your system uses eight, even if it does not.

To keep from getting confused, do not use two different names for the same variable, even if the names are equivalent on your system. Only one of the three equivalent identifiers displayed above should be used in any one program. There is another compelling reason for having only one identifier per variable. If you move your program to a system that has a limit greater than eight or no limit at all, then your program will still work; otherwise, you might need to rename some of the variables. Many compilers use all the characters in long identifiers.

Unfortunately, there is little uniformity in how systems treat upper- and lowercase letters in identifiers. Lowercase letters, if available, may or may not be treated as distinct from uppercase letters when they occur in identifiers. On some systems, the identifiers SUM, Sum, and sum are all equivalent. On other systems, they are three distinct identifiers. Needless to say, under these circumstances it is not a good idea to rely on the distinction between upper- and lowercase letters when choosing identifiers. Even if your system treats sum and SUM as distinct, there is little reason to think some other system will. Therefore, it is best to use only one of the two variants. These remarks also apply to upper- and lowercase letters in identifiers like *program*, *begin*, *integer*, and so forth, which have a standard meaning in the Pascal lan-

*names for  
variables*

*long  
identifiers*

*portability*

*upper-  
and  
lowercase*



guage. Some systems require that they always be typed in uppercase or always be typed in lowercase. To find out how your compiler treats upper- and lowercase letters, check the documentation or do a bit of experimenting.

Upper- and lowercase letters are always treated as being different when they appear in quoted strings or as the values of variables of type `char`.

This book uses a convention for upper- and lowercase letters that is designed to emphasize certain concepts: All identifiers whose meaning is defined by the Pascal language are written using all lowercase letters; all identifiers that the programmer must make up are written using at least one uppercase letter, for example, `SUM` and `TheAnswer`. We never use two spellings that differ only in the case of some or all of their letters.

*reserved  
words*

There is a special class of identifiers, called *reserved words*, that have a predefined meaning in the Pascal language and cannot be used as names for anything else, such as variables. The following is a complete list of all the reserved words we have seen so far:

*begin end program var*

In this book all reserved words are written in the special typeface shown here. A complete list of reserved words is given opposite the back cover.

*standard  
identifiers*

You may wonder why the other words that we defined as part of the Pascal language are not on the list of reserved words. What about words like `integer`, `char`, and `readln`? The answer is that, although they have a predefined meaning, you are allowed to change their meaning. Such identifiers are called *standard identifiers* and, in this book, are written using all lowercase letters, in a slightly different typeface than that used for reserved words. Needless to say, using standard identifiers as names for things other than their standard meaning can be confusing and dangerous and should thus be avoided. The safest and easiest practice is to treat standard identifiers as if they were reserved words.

---

## Putting the Pieces Together

You now know enough details about Pascal to write a program. All you need to do is put the pieces together in the right order and with the correct punctuation.

*program  
heading*

Pascal programs start with a line called the *program heading*. This consists of the reserved word *program* followed by an identifier that serves as the name of the program, followed by the standard identifiers `input` and `output`, which are separated by a comma and enclosed in parentheses. Finally, the heading is terminated with a semicolon. For example, if the program name is `Arthur`, the heading would be

*program Arthur(input, output);*

*body*

After the program heading come the variable declarations. Next comes the *body* of the program. The body of the program consists of a list of statements that serve as the instructions to be followed. This is the algorithm part of the program, and it is set off by the reserved words *begin* and *end*. The statements are separated by semicolons. As has already been pointed out, there is no need for a semicolon after the last statement because there is no other statement from which it needs to be separated. However,

---

if you do insert an extra semicolon there, it will not cause a problem. Pascal programs end with a period, placed after the identifier *end*.

In Pascal any two identifiers or numbers must be separated by one or more spaces. Hence, the program heading given above cannot start out *programArthur*. However, Pascal compilers will accept any number of extra blanks between identifiers. If two identifiers or numbers are separated by a line break or a punctuation symbol (such as a comma, semicolon, or colon), then spaces are allowed but not required.

*spacing*

Pascal allows programmers wide latitude in deciding when to start a new line. Two or more statements may be placed on the same line. With one exception, a line may be broken anywhere that a blank is allowed. The one exception is that you cannot break a quoted string across two lines. Therefore, the following is not allowed:

*line  
breaks*

```
writeln('You may NOT break a quote
across two lines like this.')
```

Almost any pattern of spacing and line breaks will be acceptable to the compiler. However, as we point out later in this chapter, programs should always be arranged so as to make them easy to read.

## TURBO Pascal

TURBO Pascal is more than just the TURBO Pascal language. It is an entire programming environment that includes its own editor and its own facilities for file handling. The TURBO editor and method of handling files are discussed in Appendixes 7, 8, and 10. In this section we will begin our discussion of the TURBO Pascal language. This section can be read without knowing the material in the appendixes, but if you are using a TURBO Pascal system, it would be best to read at least Appendix 7 before going on.

The TURBO Pascal language has more features than standard Pascal, but otherwise the two versions of Pascal are very similar. In this section we will point out a few of the differences between TURBO Pascal and standard Pascal. Sections such as this one have been added throughout the text whenever TURBO Pascal has features that differ from or extend the standard Pascal features under discussion. Readers working in standard Pascal can omit these sections entirely.

One difference between TURBO Pascal and standard Pascal is basically cosmetic. In TURBO Pascal it is not necessary to include the words *input* and *output* in the program heading. So, for example, if the first line of a standard Pascal program is

```
program Sample(input, output);
```

then the following will do as the first line of the equivalent TURBO Pascal program:

```
program Sample;
```

It does no harm to include the identifiers *input* and *output* in a TURBO Pascal program. The compiler ignores them if they are present. Thus, either of the preceding lines are acceptable and are equivalent in a TURBO Pascal program.



Another difference between TURBO Pascal and standard Pascal is also cosmetic, but of more import. In TURBO Pascal an identifier may contain the underscore symbol `_`. In TURBO Pascal the following are acceptable identifiers:

```
The_Number  N_1  N_2  Speed_Limit
```

There is no distinction between upper- and lowercase letters in TURBO Pascal. For example, in TURBO Pascal the following three identifiers are considered to be the same:

```
RATE  Rate  rate
```

*identifier  
length*

In TURBO Pascal identifiers may be of any length. (Only the first 63 characters are significant, but identifiers are seldom that long.) Hence, in TURBO Pascal, the following identifiers are considered to be three *different* identifiers:

```
TheFinalAnswer  TheFinalResult  TheFinal
```

All the programs we have seen so far work for both TURBO Pascal and standard Pascal systems. For the few programs for which this is not true, we will note that the program works only on one or the other of the two systems.

*string  
variables*

Unlike standard Pascal, TURBO Pascal allows variables of the type *string*. A value of type *string* is a sequence of characters such as

```
'Surf 's Up'  'I love you!'  '%%$12134 # Amen'
```

These are exactly the quoted strings which we discussed earlier in this chapter. The extra feature of TURBO Pascal is that a TURBO Pascal program may contain variables whose values are strings of this form.

When declaring variables to be of type *string* in TURBO Pascal, you should specify the maximum number of symbols that the string value may contain. This is done as indicated in the following example:

```
var FirstName, LastName: string[10];
```

In the above example, the two variables `FirstName` and `LastName` are declared so that they can hold string values of length ten characters or shorter. Any value from 1 to 255 may be used in place of 10 in the declaration and will allow for strings of longer or shorter maximum length.

If you omit the number and square brackets in a string variable declaration, that is equivalent to declaring the variable to hold 255 characters. For example, the declaration

```
var LongName: string;
```

is equivalent to

```
var LongName: string[255];
```

At first glance it may seem safest to declare all string variables to be of type *string* [255]. They will then have as much capacity as is possible and you can even use the shorter notation *string* in place of *string*[255]. This reasoning is correct as far as it goes. However, using a value, such as 255, which is larger than the maximum

---



### Program

```

program TURBO;
var FirstName, LastName: string[10];
    FullName: string[21];
begin
  writeln('ENTER your FIRST NAME then press RETURN');
  readln(FirstName);
  writeln('ENTER your LAST NAME then press RETURN');
  readln(LastName);
  FullName := FirstName + ' ' + LastName;
  writeln('Good day ', FullName)
end.

```

### Sample Dialogue

```

ENTER your FIRST NAME then press RETURN
Zaphod
ENTER your LAST NAME then press RETURN
Beeblebrox
Good day Zaphod Beeblebrox

```

**Figure 2.7**  
**TURBO Pascal but**  
**not standard**  
**Pascal.**

number of characters expected, will waste storage and could cause the program to run slower or fail to run at all for lack of sufficient storage.

String variables may be given a value in the same ways as variables of other types. They may be given a value by means of an assignment operator or by being read in from the keyboard using a read or readln statement. A sample TURBO Pascal program to illustrate this is given in Figure 2.7.

As illustrated in Figure 2.7, strings may be joined together using the plus sign. For example, the expression

```
'Do ' + 'Be' + 'Do'
```

returns the value 'Do BeDo'. The peculiar spacing in this result illustrates the fact that blanks are significant in strings.

These expressions which join strings with a plus sign may be used in a writeln statement, on the right-hand side of an assignment operator, or anywhere else that a value of a *string* type may be used. For example, suppose the following line occurs in a program:

```
writeln('Do ' + 'Be' + 'Do');
```

When this line is executed the following will appear on the screen:

```
Do BeDo
```

There is an alternative, equivalent notation that may be used instead of the plus sign when joining strings. For example, instead of

```
'Do ' + 'Be' + 'Do'
```

you may use the following and it will have the exact same meaning:

```
concat('Do ', 'Be', 'Do')
```

Joining strings together is called *concatenating* the strings, which will explain the origin of the notation `concat`. Which notation you use is purely a matter of taste. The program in Figure 2.8 illustrates the `concat` notation. The program in Figure 2.8 could have been written using plus signs instead of `concat`. The program in Figure 2.7 could have been written using `concat` instead of plus signs.

The program in Figure 2.8 also illustrates how you may refer to individual charac-

### Program

```
program Namer;
var FirstName, LastName: string[10];
    FullName, ShortName: string[21];
    Initials: string[5];
begin
  writeln('This program produces shortened versions of names. ');
  writeln('Enter your first name and then press return: ');
  readln(FirstName);
  writeln('Enter your last name and then press return: ');
  readln(LastName);

  FullName := concat(FirstName, ' ', LastName);
  ShortName := concat(FirstName[1], '.', LastName);
  Initials := concat(FirstName[1], '.', LastName[1], '.');

  writeln('Good day ', FullName);
  writeln('Here are some shortened versions of your name: ');
  writeln(ShortName);
  writeln(Initials);
  writeln('Goodbye ', Initials)
end.
```

### Sample Dialogue

This program produces shortened versions of names.

Enter your first name and then press return:

**Zaphod**

Enter your last name and then press return:

**Beeblebrox**

Good day Zaphod Beeblebrox

Here are some shortened versions of your name:

Z. Beeblebrox

Z. B.

Goodbye Z. B.

**Figure 2.8**  
**Another TURBO**  
**Pascal program.**

ters within a string variable. Before we discuss how this is done in that program, we will first discuss the general principles underlying its use.

If `Word` is a variable of a string type, then `Word[1]` denotes the first character of the string value of `Word`, `Word[2]` denotes the second character, and so forth. For example, suppose that the *string* variable `Word` is declared by

```
var Word: string[10];
```

and suppose the program contains the following assignment statement.

```
Word := 'abc'
```

After this assignment, the value of `Word[1]` will be 'a', the value of `Word[2]` will be 'b', the value of `Word[3]` will be 'c'. Since the value of `Word` has only three characters, the expression `Word[4]` has no meaning and should not be used.

The use of this notation is illustrated in Figure 2.8. In the program in Figure 2.8, the string variable `FirstName` is set equal to 'Zaphod' (by a `readln` statement). Since `FirstName` has value 'Zaphod', `FirstName[1]` has value 'Z'.

Additional string manipulating features of TURBO Pascal are discussed in Chapter 8.

For some purposes that we will encounter later in this text, the two variables declared below are considered to be of different types:

```
var Short: string[10];  
    Long: string[50];
```

However, it is perfectly acceptable to mix such variables in assignment statements. The following is allowed and has the obvious meaning:

```
Long := Short
```

The following is also allowed. However, if the value of `Long` is more than ten characters in length, then only the first ten characters will be assigned to `Short`.

```
Short := Long
```

In situations like that above, we will often refer to variables such as `Long` and `Short` as variables of *the type string* without mentioning any numbers in square brackets. However, the number in square brackets should appear in the variable declaration.

The type `char` and the *string* types are compatible in only one direction. You can use a value of type `char` where a value of a *string* type is expected, but not the other way around. If `Letter` is a variable of type `char` and `Word` is a variable of a *string* type, then the following is allowed and has the obvious meaning:

```
Word := Letter
```

However, the reverse is illegal. The following is illegal, whenever `Letter` is of type `char` and `Word` is of any *string* type:

```
Letter := Word {ILLEGAL}
```

The above is illegal even if `Word` is of type *string*[1].

If variables of type *string* are not used with care, they can present some problems with input. To illustrate a problem that can arise, suppose `Name` is declared to be



of type *string*[20], suppose Age is of type integer, and consider the following pair of statements:

```
writeln('ENTER your NAME and AGE');
readln(Name, Age)
```

If, in response to the prompt line, the user types in the following on one line,

**Ursa Minor 550**

and then presses the return key, the variable Name will get the value

'Ursa Minor 550'

Remember, blanks and digits are characters and so can be part of a value of type *string*. A safer sequence of statements is

```
writeln('ENTER your NAME and press RETURN');
readln(Name);
writeln('ENTER your AGE and press RETURN');
readln(Age)
```

---

## Introduction to Programming Style

All the sample programs we have seen were laid out in a particular format. For example, the statements were all indented by the same amount. Similarly, the declarations were aligned. These and other matters of style are of more than aesthetic interest. A program that is written with careful attention to style is easier to read, easier to correct if it contains a mistake, and easier to change should that prove desirable at some later time.

*indenting*

*blank  
lines*

A program should be laid out so that elements that are naturally thought of as a group look like a unit. The standard way of doing this is to indent everything in that group by the same amount. Another way to make a program more readable is to skip a line between pieces that are logically thought of as separate. The important point is to make separations using indentations and line breaks. The exact number of spaces in an indentation is a matter of personal taste. Sometimes there are also natural break indicators, such as the words *begin* and *end*, which can be made to stand out and frame a group.

*choosing  
names*

Variables, constants, and even program names should at least hint at their meaning or use. It is easy just to use X, Y, and Z again and again as variables for numbers. However, it is much easier to understand a program if the variables have meaningful names. Contrast

```
X := Y * Z
```

with the more suggestive statement below:

```
Pay := Rate * Hours
```

The two statements accomplish the same thing, but the second is easier to understand.

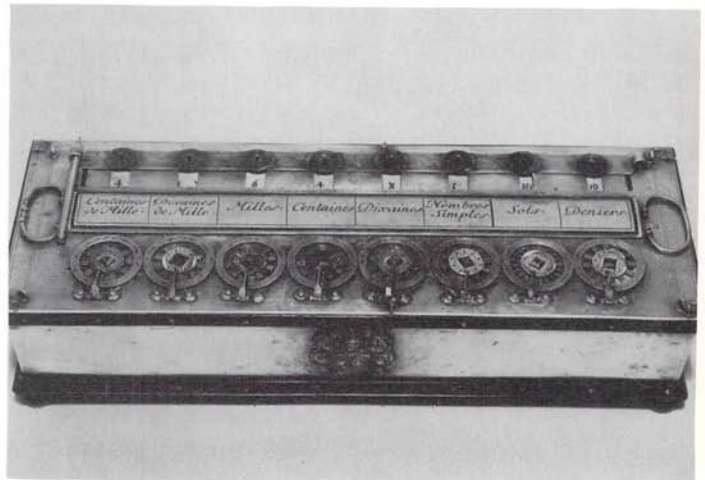
---

## Blaise Pascal

(Optional)

The language Pascal was named after Blaise Pascal, a mathematician, engineer, scientist, and religious philosopher. Pascal was born in 1623 in Auvergne in central France. At the age of 18, he designed a computing machine capable of performing simple arithmetic calculations. The machine was a type of adding machine and not the sort of programmable machine that would today be called a computer. Nonetheless, the machine received a good deal of attention and served as a prototype for a number of later computing machines. Pascal had a number of models built and attempted to market the machine. Due to its high price, however, the machine was never a financial success.

His calculating machine was only one of Pascal's numerous scientific and engineering contributions. He designed Paris's first public bus system, which used horse-drawn carriages. He made important contributions to geometry, probability theory, and hydrodynamics. Pascal was also a prominent figure in religious philosophy. He belonged to a controversial movement within the Catholic church known as Jansenism. His last and most widely read religious work is now published under the title *Pensées*. The quotations of Pascal in this book were taken from that work. Although the book was intended as a work on religious philosophy, some of the remarks do seem to apply to the philosophy of computer programming.



**Blaise Pascal and  
his calculating  
machine**

## Self-Test Exercises

1. What is the output produced by the following three lines (when correctly embedded in a complete program)? The variables are of type `integer`:

```
X := 2; Y := 3;
Y := X;
writeln(X, Y)
```

2. What is the output produced by the following three lines (when correctly embedded in a complete program)? The variables are of type `integer`:

```
X := 2;
X := X + 1;
writeln(X)
```

3. What is the output produced by the following two lines (when correctly embedded in a complete program)? The variables are of type `char`:

```
A := 'B'; B := 'C'; C := A;
writeln(A, B, C, 'C')
```

4. Which of the following are correctly formed constants of type `integer`?

3.5    4.0    4.    4    1,295    9/3    8/5    '7'

5. Which of the following are correctly formed constants of type `real`?

98.6    -33.4    .89    -.89    3,987.85    4.    4    4.0

6. Which of the following are correctly formed constants of type `real`?

.57E12    57E12    57E-12    57E3.7    57.9E3.7    -9.8E2

7. Convert each of the following (non-Pascal) arithmetic expressions into Pascal arithmetic expressions:

$$3x \quad 3x + y \quad \frac{x + y}{7} \quad \frac{3x + y}{z + 2}$$

8. What (if anything) is wrong with the following declarations?

- (a) `var Count: integer ;`  
`Answer : char; AMOUNT: integer;`
- (b) `var Time: integer;`  
`var Rate: real;`
- (c) `var Count1; Count2: integer;`  
`Rate: real;`
- (d) `var N1, N2: integer;`  
`AVE: real`

9. What are the types of the values of the following expressions? (You need not evaluate them.)



2 \* 3    5 \* (7 + 4/2)    '3'  
 2.0000    2E9    2/3

10. The following program contains errors. What are they?

```
program Test
begin
  writeln("Hello");
  writeln('This program was written in a hurry');
  writeln('It contains a few mistakes');
  writeln('Can you find them?')
  writeln('The compiler can')
end
```

### Interactive Exercises

11. Type up and run the following program:

```
program DoMeFirst(input, output);
begin
  writeln('Hello. ');
  writeln('End of program')
end.
```

12. Write a program that reads in one integer, multiplies it by 2, and then writes the result back to the screen.

---

“Don’t stand chattering to yourself like that,” Humpty Dumpty said, looking at her for the first time, “but tell me your name and your business.”

“My *name* is Alice, but—”

“It’s a stupid name enough!” Humpty Dumpty interrupted impatiently. “What does it mean?”

“*Must* a name mean something?” Alice asked doubtfully.

“Of course it must,” Humpty Dumpty said with a short laugh: “*my* name means the shape I am—and a good handsome shape it is, too. With a name like yours you might be any shape, almost.”

*Lewis Carroll, Through the Looking-Glass*

---



---

## Problem Solving and Program Design

To keep from getting confused when designing a program and to produce a readable, easy-to-change program require patience and a systematic approach to the design pro-

---

cess. In the first chapter we outlined one such systematic approach. It consisted of carefully analyzing the problem, designing an algorithm for a hypothetical person to follow, and then translating the algorithm into a Pascal program for the computer to follow. One basic design technique for producing the algorithm is called *top-down design*. We next introduce this technique and illustrate it with a design example.

---

## Top-Down Design

*stepwise  
refinement*

A good plan of attack for designing an algorithm is to break down the task to be accomplished into a few big subtasks, then decompose each big subtask into smaller subtasks, then replace the smaller subtasks by even smaller subtasks, and so forth. Eventually the subtasks become so small that they are trivial to implement in Pascal or whatever language you are using. This method is usually called *stepwise refinement* or *top-down design* or, more graphically, *divide and conquer*.

Not only is stepwise refinement an efficient design method, it also produces a good algorithm in the sense that the algorithm is easier to understand and subsequent modifications are relatively easy to make. This is very important as most programs are changed at some time, and some of them are being changed constantly. For example, a simple computerized airline reservation system might be expanded to keep track of seat as well as flight reservations. The top-down design method is illustrated in the two case studies in this chapter.

---

## Case Study

---

### A Guessing Game

#### Problem Definition

We want to design a program to play a simple game with the user. The rules of the game are as follows: The user chooses two numbers and then tries to guess their average. The user could calculate the average with pencil and paper, but the idea is to choose relatively large, complicated numbers and to really guess rather than compute the average. The program lets the user know whether the guess was correct. This is not much of a game; nonetheless, it can be used to illustrate the program-design process.

#### Discussion

*subtasks*

*begin*

1. Have the user input two numbers and a guess of their average.
2. Calculate the average.
3. Output enough information to permit the user to tell whether the guess is correct.

*end.*

---

An algorithm for the first subtask is the following:

*ALGORITHM*

- 1a. Ask the user to type in two numbers;
- 1b. `readln(N1, N2);`
- 1c. Ask the user to enter a guess as to their average;
- 1d. `readln(GUESS)`

Our algorithm for subtask 1 contains a mixture of Pascal and English. This is quite common. When the Pascal way to express a step is obvious, there is little point in writing it in English. When the steps are large or complicated, they are usually first expressed in English. This combination of English and Pascal is sometimes called *pseudocode*.

*pseudocode*

The algorithm for subtask 1 translates into the following Pascal code:

```
writeln('Enter TWO INTEGERS, separated by a SPACE');
writeln('Then press RETURN');
readln(N1, N2);
writeln('Now GUESS their AVERAGE, ');
writeln('Enter your GUESS, then press RETURN');
readln(GUESS)
```

*implementation  
phase*

The second task is to compute the average of the numbers held in the variables N1 and N2. The definition of average yields our algorithm:

*ALGORITHM*

- 2a. Compute the sum of N1 and N2;
- 2b. Divide the sum by 2 to get the average

In order to translate this, we use another variable of type `real` to hold the average. If we call the variable AVE, we get the following Pascal code for subtask 2:

```
AVE := (N1 + N2) / 2
```

Designing step 3 requires some thought. Since we are not yet experienced programmers, we will settle for a very simpleminded solution. The computer will announce the user's guess and the correct average. The user can then see if the guess matches the true average. This solution is easy enough to translate directly into Pascal:

```
writeln('You guessed ', GUESS);
writeln('The right answer is ', AVE)
```

The complete program is shown in Figure 2.9. A blank line separates the code for each of the three major subtasks from one another.

---

## Integer Division—mod and div

There is a version of division that applies only to values of type `integer` and that returns values of type `integer`. It is essentially the “long division” you learned in grade school. For example, 17 divided by 5 yields 3 with a remainder of 2. The two numbers obtained in this way can be produced with the Pascal operators *div* and *mod*. The *div* operation yields the number of times one number “goes into” another.

---



**Program**

```

program Game(input, output);
var N1, N2: integer;
    GUESS, AVE: real;
begin
    writeln('Enter TWO INTEGERS, separated by a SPACE');
    writeln('Then press RETURN');
    readln(N1, N2);
    writeln('Now GUESS their AVERAGE, ');
    writeln('Enter your GUESS, then press RETURN');
    readln(GUESS);

    AVE := (N1 + N2)/2;

    writeln('You guessed ', GUESS);
    writeln('The right answer is ', AVE)
end.

```

**Sample Dialogue**

```

Enter TWO INTEGERS, separated by a SPACE
Then press RETURN
56 99
Now GUESS their AVERAGE,
Enter your GUESS, then press RETURN
76.5
You guessed    76.50
The right answer    is 77.50

```

**Figure 2.9**  
Game-playing  
program.

The *mod* operation gives the remainder. The operator *mod* is short for “modulo.” In both cases, the divisor is given second. For example, the statements

```

writeln('17 Divided by 5 is ', 17 div 5);
writeln('with a Remainder of ', 17 mod 5);

```

yield the following output:

```

17 Divided by 5 is 3
with a Remainder of 2

```

Figure 2.10 shows the relationship between these operations and grade-school long division. Figure 2.11 illustrates the differences between the two kinds of division in Pascal.

**Figure 2.10**  
*mod* and *div*.

$\begin{array}{r} 4 \\ 3 \overline{)12} \\ \underline{12} \\ 0 \end{array}$	$\xrightarrow{\quad} 12 \text{ div } 3$	$\begin{array}{r} 4 \\ 3 \overline{)14} \\ \underline{12} \\ 2 \end{array}$	$\xrightarrow{\quad} 14 \text{ div } 3$
	$\xrightarrow{\quad} 12 \text{ mod } 3$		$\xrightarrow{\quad} 14 \text{ mod } 3$

Expression	Value	Expression	Value
16 <i>div</i> 5	3	17 <i>div</i> 5	3
16 <i>mod</i> 5	1	17 <i>mod</i> 5	2
16/5	$(3 + 1/5 =) 3.2$	17 / 5	$(3 + 2/5 =) 3.4$

**Figure 2.11**  
The different kinds  
of division.

---

## Desktop Testing

We tend to assume that the person or machine that is following our instructions will not do anything “stupid” and will fill in any “obvious” detail. Both of these assumptions are incorrect when applied to computers. The computer does exactly what the program’s instructions say, and in our design strategy, these instructions are simply translations of the instructions in an algorithm expressed in English or pseudocode. Hence, you should test the algorithm before you attempt to translate it into a Pascal program. The testing can be done by mentally stepping through the algorithm and carrying out the instructions. By stepping through an algorithm you can see it in operation and can detect many mistakes in detail as well as find any missing details. When doing this, you will want to use a pencil and paper to write down the values of the variables and to keep track of how the values change.

---

## Case Study

---

### Making Change

In this section we will design a sample program to make change. Given an amount of money, the program will tell how many of each type of coin, such as quarters, dimes and so forth, it takes to equal the given amount.

#### Problem Definition

The statement of the task to be accomplished by the program seems pretty clear, but before we go on let us make sure that as few details as possible are left unspecified. For example, what is the range of inputs for which the program must work? Does it need to give out dollars as change or not? If yes, what denominations of bills should it use? What should it do with an input of zero? The answers depend on the use to which the program is to be put.

For this example, we will assume that the program is to be used by a cashier who has no trouble with dollars but needs to be told what coins to hand out and who knows that zero cents means no coins. Hence, we can assume that the amount of change will

---

range from one to ninety-nine cents. Are there any other points left unspecified? What if the cashier runs out of a particular coin? After all, this cashier is not too smart. That is why we were asked to write the program. We inquire and discover that the cashier usually has very few if any half-dollar coins and occasionally runs out of nickels but never runs out of any other coin. After consulting with the cashier's boss, we decide to ignore half-dollar coins and nickels. These coins will just be brought to the bank at the end of the day.

### Discussion

Now that we understand the problem, we can start to design an algorithm. Our first attempt is the following:

*begin*

1. Input the amount;
2. Compute a combination of quarters, dimes, and pennies whose value equals the amount;
3. Output the list of coins

*end.*

We have broken our task down into three subtasks. Now we must solve these subtasks and produce some Pascal code for the solutions. (In this context, the word *code* means a part of a program.)

The first subtask is easy. We simply read the amount into a variable. To make our program easy to read, we choose the name `Amount` for this variable. Subtask 1 is accomplished by the following:

```
writeln('Enter an amount of change');  
writeln('from 1 to 99 cents: ');  
readln(Amount)
```

### ALGORITHM refinement

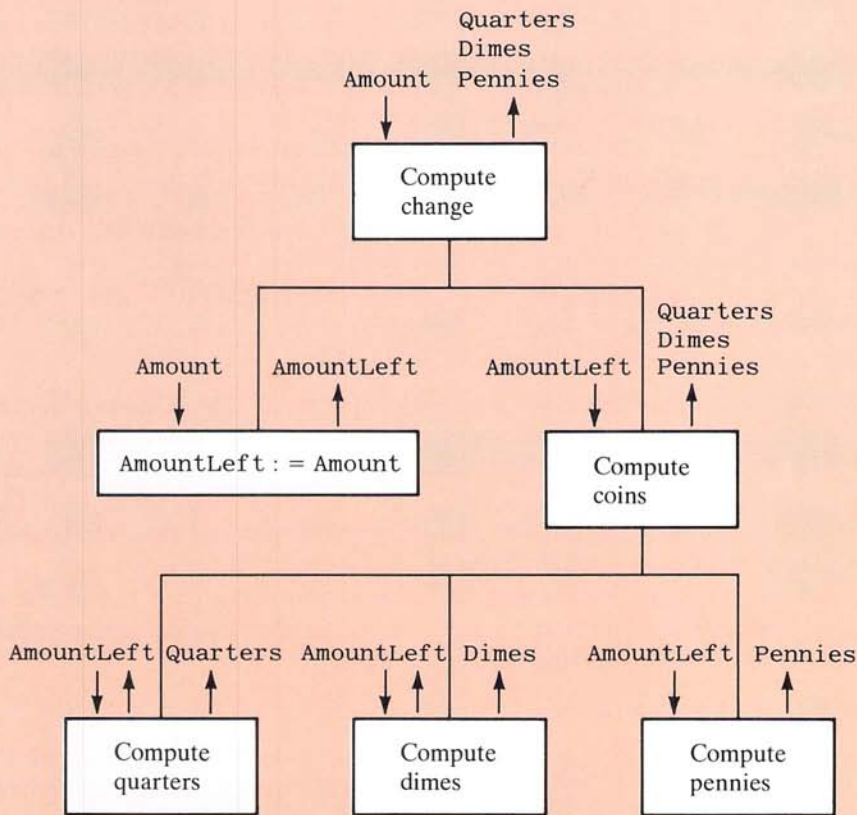
The second subtask is still quite large and it will help to break it down into still smaller tasks. One sensible breakdown is:

- 2a. Compute the number of quarters to give out;
- 2b. Compute the number of dimes to give out;
- 2c. Compute the number of pennies to give out

These subtasks are not completely independent. For example, the number of dimes given out will depend on the number of quarters given out. In other words, the number of dimes will be computed on the basis of the original amount minus the total value of the quarters given out. When we analyze the information that is being passed between subtasks, we see that there are two different notions of amount. There is the original amount that the cashier input. We decided to save this amount in the variable called `Amount`. We will also need a variable to store the amount left to be given out as we calculate the number of quarters, dimes, and pennies. We might be tempted to use the variable `Amount` for this purpose by decreasing it successively by the values of quarters and dimes. However, we would like to output the original amount when we output the numbers of coins, and so we wish to preserve the value of `Amount` unchanged. We

---





### Data Summary

**Amount:** the amount to be given as change.

**AmountLeft:** used to keep track of the amount not yet calculated as some type of coins.

**Quarters, Dimes, Pennies:** the numbers of coins to be given out.

**Figure 2.12**  
Data flow diagram  
for change  
program.

therefore introduce a new variable called `AmountLeft`, which, at various points in the computation, will hold the amount still left to be given as coins. At the start of the computation to calculate coins, the entire amount is the amount left. Hence, we will insert the following assignment before we calculate the number of coins.

```
AmountLeft := Amount
```

After the program calculates the number of quarters to give out, `AmountLeft` will be decreased by the value of the quarters given out. After computing the number of dimes, `AmountLeft` will be further decreased by the value of the dimes. The movement of information such as the value of `AmountLeft` from one subtask to another is often called *data flow*. This interaction of subtasks is depicted in Figure 2.12. Diagrams like this will be called *data flow diagrams*.

*data flow*

### ALGORITHM second refinement

The smaller tasks, into which subtask 2 is subdivided, can now be rewritten into the following more detailed decomposition of subtask 2:

- 2a. Compute the maximum number of quarters in `AmountLeft` and decrease `AmountLeft` by the total value of the quarters;
- 2b. Compute the maximum number of dimes in `AmountLeft` and decrease `AmountLeft` by the total value of the dimes;
- 2c. Compute the number of pennies in `AmountLeft`

### analysis of possible solutions

Subtask 2a is already expressed as two smaller subtasks. The first one is to compute the maximum number of quarters in `AmountLeft`. There are a number of ways to do this. We could subtract 25 from `AmountLeft` as many times as is possible (without getting a negative result) and count the number of times we do that. That would work, but before we jump to the keyboard and type it up, we should think about whether there is a simpler solution. There is.

The number of times we can subtract 25 from `AmountLeft` is the number of times 25 “goes into” `AmountLeft`, and we have a standard Pascal operator for that. The number of times 25 goes into `AmountLeft` is:

```
AmountLeft div 25
```

This yields the number of quarters to be given out, and the value will be stored in a variable called `Quarters`. The first part of subtask 2a can therefore be accomplished by the Pascal statement:

```
Quarters := AmountLeft div 25
```

The decrease in the amount left after giving out the quarters can be computed by:

```
AmountLeft := AmountLeft - 25*Quarters
```

Although that formula will work, again there is a simpler expression for the new amount left. The amount left after giving out as many quarters as possible is just the remainder after dividing the old value of `AmountLeft` by 25. In Pascal, this is expressed by

```
AmountLeft := AmountLeft mod 25
```

Hence, subtask 2a can be accomplished by the following Pascal code:

```
Quarters := AmountLeft div 25;
AmountLeft := AmountLeft mod 25
```

Subtask 2b can be accomplished by a similar piece of Pascal code:

```
Dimes := AmountLeft div 10;
AmountLeft := AmountLeft mod 10
```

### Program

```
program Change(input, output);
var Amount, AmountLeft,
    Quarters, Dimes, Pennies: integer;
begin
    writeln('Enter an amount of change');
    writeln('from 1 to 99 cents:');
    readln(Amount);

    AmountLeft := Amount;

    Quarters := AmountLeft div 25;
    AmountLeft := AmountLeft mod 25;

    Dimes := AmountLeft div 10;
    AmountLeft := AmountLeft mod 10;

    Pennies := AmountLeft;

    writeln(Amount, ' cents can be given as:');
    writeln(Quarters, ' quarters');
    writeln(Dimes, ' dimes and');
    writeln(Pennies, ' pennies')
end.
```

### Sample Dialogue

```
Enter an amount of change
from 1 to 99 cents:
67
67 cents can be given as:
2 quarters
1 dimes and
7 pennies
```

**Figure 2.13**  
A program to  
make change.

The purpose of the variable `Dimes` above and the variable `Pennies` mentioned next is obvious.

Subtask 2c is very simple, because the entire amount remaining is given out as pennies. It is accomplished by:

```
Pennies := AmountLeft
```

Subtask 3 can be accomplished by a series of `writeln` statements that output the values of the variables `Quarters`, `Dimes`, and `Pennies`. As a check to make sure that the original amount was entered correctly, the value of `Amount` is also output. The complete program is shown in Figure 2.13.

*echoing  
input*



## Exploring the Solution Space

*alternative  
solutions*

The previous discussion about computing the number of quarters illustrates an important lesson about algorithms and their design. There is always more than one algorithm for a given task. Just because you have found one algorithm does not mean you have found the best algorithm. It always pays to look for a simpler or more efficient algorithm.

*solution by  
analogy*

The last case study illustrates another technique to help in your search for algorithms. An existing algorithm can often be adapted to fit a new task. In the case study, we used the following to compute the number of quarters and the amount left after giving out that many quarters.

```
Quarters := AmountLeft div 25;  
AmountLeft := AmountLeft mod 25
```

To accomplish a similar task for dimes, all we needed to do was to replace 25 with 10.

We will present other techniques for algorithm design in later chapters, but first we will summarize the tools we have developed thus far for designing algorithms and Pascal programs.

---

There was a most ingenious Architect who had contrived a new Method for building Houses, by beginning at the Roof, and working downwards to the Foundation;

*Jonathan Swift, Gulliver's Travels*

---

---

## Summary of Problem Solving Techniques

- Before writing a Pascal program, design the algorithm (method of solution) that the program will use. The algorithm can be expressed in a combination of Pascal and English known as *pseudocode*.
  - You should design algorithms using the top-down (divide and conquer) method described in this chapter.
  - When designing programs by the top-down method, you should explicitly analyze the flow of information between subtasks.
  - You can sometimes produce algorithms by adapting a known algorithm to fit the new task.
  - Always look for alternative solutions. Just because you have found one solution to a problem does not mean that you have found the best solution.
  - Give an algorithm a “desktop” testing before translating it into a Pascal program.
-

---

## Summary of Programming Techniques

- Both pseudocode and the final Pascal program should use meaningful names for variables.
- Use an indenting, spacing, and line-break pattern similar to the sample programs.
- All data has a data type. Be sure to check that variables and constants are of the correct data type.
- Use enough parentheses in arithmetic expressions to make the order of operations clear.
- Be sure that variables are initialized before the program attempts to use their value.
- When programming interactively, always include a prompt line in a program whenever the user is expected to enter data.
- Programs should always echo input.

---

## Summary of Pascal Constructs

Each of the following chapters contains a summary similar in form to the following one. The entries in the summary consist of three parts: (1) an outline of the syntax (form) of the construct, (2) one or more typical examples of the construct, and (3) an explanation of the construct. Occasionally, one or two of the parts are omitted.

### identifier

Syntax:

Any string of letters and digits that begins with a letter.

Example:

Sum2

Identifiers are used as names for variables and other items in programs. (TURBO Pascal allows the use of the underscore symbol in identifiers, as in, for example, Hot\_Shot\_2.)

### variable declarations

Syntax:

```
var <variable list 1>: <type 1>;
    <variable list 2>: <type 2>;
    .
    .
    .
    <variable list n>: <type n>;
```

Example:

```
var N1, N2: integer;
    Rate: real;
```

---

Each variable list is a list of identifiers separated by commas. All the identifiers on <variable list 1> are declared to name variables of type <type 1>, all the identifiers on <variable list 2> are declared to name variables of type <type 2>, and so forth. The types may be any type names. They may be in any order and may be repeated. The types we have seen so far are `integer` for whole number values, `real` for number values with a fractional part, and `char` for values that are a single character. (TURBO Pascal also allows *string* types).

### the type integer

Syntax:

`integer`

The data type whose values are all the whole numbers (positive, negative, or zero) that the computer system can handle. Constants of this type are written as strings of digits optionally preceded by a plus or a minus sign. The values are stored exactly.

### the type real

Syntax:

`real`

The data type whose values are numbers that, when written in the usual decimal notation, have digits after the decimal point. The values are stored as approximate values. Constants of this type may be in either of the following forms, optionally preceded by a plus or a minus sign:

1. A sequence of digits containing a decimal point that has at least one digit before the decimal point and at least one digit after the decimal point.
2. A number followed by the letter E, followed by a constant of type `integer`. The number before the E must be either an `integer` constant or a `real` constant in form 1.

### the type char (characters)

Syntax:

`char`

The type consisting of single characters. The constants are formed by placing the character in single quotes. For example, `'A'`, `'$'`, and `'3'`.

### assignment statement

Syntax:

`<variable> := <expression>`

Example:

`SUM := N1 + N2`

The <expression> is evaluated and the value of the <variable> is set to that value. The <variable> and the value of the <expression> must be of the same type.

---



### integer division

Syntax:

```
<integer expression 1> div <integer expression 2>  
<integer expression 1> mod <integer expression 2>
```

Examples:

```
X := 14 div 3;  
Y := 14 mod 3
```

*div* returns the quotient obtained from dividing the value of the first expression by the value of the second; *mod* returns the remainder. In the examples, the value of X is changed to 4 and that of Y is changed to 2.

### write statement

Syntax:

```
write(<argument list>)
```

Example:

```
write(' Answer is ', SUM)
```

Outputs the values of the items in <argument list> to the primary output device, usually a display screen. <argument list> is a list of variables and quoted strings separated by commas.

### quoted strings

Example:

```
'Surf''s up'
```

When used in a `writeln`, the string inside the single quotes is output to the screen. Both the opening and closing quotes are the same symbol. To include a single quote within a quoted string, you must use two single quotes. Quoted strings may not be broken across two lines.

### write-line statement

Syntax:

```
writeln(<argument list>)
```

Example:

```
writeln(' Answer is ', SUM)
```

Same as `write` except that any subsequent output will appear on a new line.

### read statement

Syntax:

```
read(<variable list>)
```

Example:

```
read (N1, N2)
```

The `<variable list>` is a list of  $n$  variables separated by commas. There may be any number  $n$  of variables. This statement causes the computer to read  $n$  values from the primary input device, usually the keyboard, and to set the values of the variables in `<variable list>` to these values. The values read must correspond in type to the variable types.

### read-line statement

Syntax:

```
readln (<variable list>)
```

Example:

```
readln (N1, N2)
```

Same as the `read` statement except that any subsequent input will be taken from the next line of input.

---

## Exercises

### Self-Test Exercises

13. Determine the value of each of the following Pascal arithmetic expressions:

<code>15 div 12</code>	<code>15 mod 12</code>
<code>24 div 12</code>	<code>24 mod 12</code>
<code>123 div 100</code>	<code>123 mod 100</code>
<code>200 div 100</code>	<code>200 mod 100</code>
<code>99 div 2</code>	<code>99 mod 2</code>
<code>2 div 3</code>	<code>2 mod 3</code>

### Interactive Exercises

14. Write a program that reads two integers into the variables `X` and `Y` and then outputs `X div Y` and `X mod Y`. Run the program several times with different pairs of integers as input.

15. Write a program that will convert a number of seconds to the equivalent number of minutes and seconds. Use the `mod` and `div` operators.

16. Type up and run the program given in Figure 2.1. Then modify the program so that it does subtraction instead of addition. Do not forget to change the string ' Plus ' to something appropriate. Run the modified program. Next, modify the program so that it does multiplication instead of addition or subtraction. Run that program.

17. Modify your program from the previous exercise so that it uses data and variables

---

of type `real` instead of type `integer`. Run the program. Modify the program again so that it does division instead of multiplication.

18. Modify your program from the previous exercise so that it outputs the equal sign instead of the word 'Equals'.

19. Write a Pascal program that will read in a character typed in on the keyboard and then write it to the screen twice.

20. Write a Pascal program that will read in two characters and then write them both out twice. Remember that the blank is a perfectly good character to the computer, and so things will go wrong if you separate the characters by a blank.

21. Type up and run the program shown in Figure 2.9.

22. Modify the program from the previous exercise so that it also outputs one additional line giving the amount by which the user's guess missed the true average. For this exercise, it is acceptable to output zero or a negative number as the amount by which the guess missed the true average.

### Programming Exercises

23. Write a Pascal program that reads in two integers and then outputs their sum, difference, and product.

24. Write a Pascal program that reads in two integers, divides one by the other, places the result in a variable of type `real`, and then outputs both the numbers and their quotient. Be sure to include a `writeln` statement that warns the user not to give input that would cause the computer to try to divide by zero.

25. A class has four exams in one term. Write a program that reads in a student's four exam scores, as integers, and outputs the student's average.

26. A metric ton is 35,273.92 ounces. Write a program that reads in the weight of a package of breakfast cereal in ounces and then outputs the weight in metric tons as well as the number of boxes of cereal needed to yield one metric ton of cereal.

27. A government research lab has concluded that certain chemicals commonly used in foods will cause death in laboratory mice. A friend of yours is desperate to lose weight but cannot give up soda pop. Your friend wants to know how much diet soda pop it is possible to drink without dying as a result. Write a program to supply the answer. The input to the program is the amount of artificial sweetener needed to kill a mouse, the weight of the mouse, and the weight of the dieter. To ensure the safety of your friend, be sure the program requests the weight at which the dieter will stop dieting, rather than the dieter's current weight. Assume that diet soda contains one-tenth of 1% artificial sweetener.

28. A Celsius (centigrade) temperature  $C$  can be converted to an equivalent Fahrenheit temperature  $F$  according to the following formula:

$$F = (9/5)C + 32$$

Write a Pascal program that reads in a Celsius temperature as a decimal number and then outputs the equivalent Fahrenheit temperature.



29. The straight-line method for computing the yearly depreciation in value  $D$  for an item is given by the formula

$$D = \frac{P - S}{Y}$$

where  $P$  is the purchase price,  $S$  is the salvage value, and  $Y$  is the number of years the item is used. Write a program that takes as input the purchase price of an item, its expected number of years of service, and its expected salvage value and then outputs the yearly depreciation for the item.

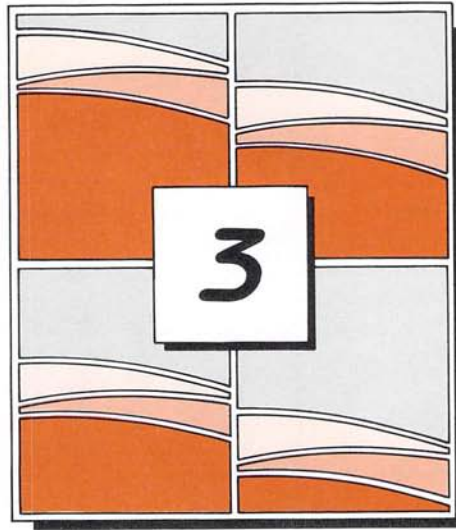
30. An automobile is used for commuting purposes. Write a program that takes as input the distance of the commute, the automobile's fuel efficiency in miles per gallon, and the price of gasoline and then outputs the cost of gasoline for the commute.

31. Workers at a particular company have won a 7.6% pay increase. Moreover, the increase is retroactive for six months. Write a program that takes an employee's previous annual salary as input and then outputs the amount of retroactive pay due the employee, the new annual salary, and the new monthly salary.

32. The public utilities commission has decided that the electric company overcharged its customers for two months last year. To make up the difference to the customers, the commission orders the company to decrease each of next month's bills by 10%. The city also levies a 3% utility tax, which is to be applied to the bill before it is discounted. Also, the 10% discount does not apply to the utility tax. Assume electricity costs \$0.16 per kilowatt-hour. Write a program to compute next month's electricity bill given the number of kilowatt-hours consumed as input.

33. Write a Pascal program for the algorithm about discount installment loans that was given as Exercise 10 in Chapter 1.

---



## *More Pascal and Programming Techniques*

“ . . .—and that shows that there are three hundred and sixty-four days when you might get un-birthday presents—”

“Certainly,” said Alice.

“And only *one* for birthday presents, you know. There’s glory for you!”

“I don’t know what you mean by ‘glory,’” Alice said.

Humpty Dumpty smiled contemptuously. “Of course you don’t —till I tell you. I meant ‘there’s a nice knock-down argument for you!’”

“But ‘glory’ doesn’t mean ‘a nice knockdown argument,’” Alice objected.

“When *I* use a word,” Humpty Dumpty said, in rather a scornful tone, “it means just what I choose it to mean—neither more nor less.”

“The question is,” said Alice, “whether you *can* make words mean so many different things.”

“The question is,” said Humpty Dumpty, “which is to be master—that’s all.”

*Lewis Carroll, Through the Looking-Glass*

## Chapter Contents

Naming Constants	Optional Else
Comments	Pitfall—Extra Semicolons
Pitfall—Forgetting a Closing Comment Delimiter	Compound Statements
Formatted Output	Iterative Enhancement
Example Using Named Constants and Formatted Output	Case Study—Payroll Calculation
Allowable Range for Numbers	Standard Functions
More about Commenting	Using Known Algorithms
Testing and Debugging	Case Study—Solving Quadratic Equations
Tracing	Defensive Programming
Use of Assertions in Testing (Optional)	More about Indenting and Commenting
Self-Test Exercises	Summary of Problem Solving Techniques
Interactive Exercises	Summary of Programming Techniques
Syntax Diagrams	Summary of Pascal Constructs
Simple Branching—If-Then-Else	Exercises
Pitfall—The Equality Operator	References for Further Reading

**I**n this chapter we present more features of the Pascal language, including a mechanism that allows Pascal programs to choose between alternative actions. We develop two more sample programs, from problem formulation through to Pascal program. In the process we illustrate some new problem solving techniques. We also present some key programming techniques, including techniques for testing programs and correcting programming errors. Since programming style is of more than just aesthetic importance, this chapter both opens and closes with remarks on style. The first topic is a Pascal construct that is used to make programs more readable and easier to modify.



---

## Naming Constants

There are two problems with constants in a computer program. The first is that they carry no mnemonic value. For example, when the number 10 is encountered in a program, the number gives no hint of its significance. If the program is a banking program, it might be the number of branch offices or the number of teller windows at the main office. In order to understand the program, you need to know the significance of each constant. The second problem is that when a program needs to be changed, the process of changing constants tends to introduce errors. Suppose that 10 occurs 12 times in a banking program; four times it represents the number of branch offices, and eight times it represents the number of teller windows at the main office. When the bank opens a new branch and the program needs to be updated, there is a good chance that some of the 10's that should be changed to 11 will not be, or that some that should not be changed to 11 will be changed. Pascal provides a single mechanism to deal with all of these problems.

In Pascal you can assign a name to a constant and then use the name in place of the constant. This is done with a *constant declaration*. Constant declarations are placed between the program heading and the variable declarations.

*constant  
declarations*

A constant declaration consists of the reserved word *const* followed by the identifier that is to be the name of the constant, followed by the equal sign and then the constant. The declaration includes sufficient blanks to separate the various pieces and is ended with a semicolon. In this case, an example is clearer than a definition. The following gives the name `BranchCount` to the number 10:

```
const BranchCount = 10;
```

To declare more than one constant, simply list them all, separated by semicolons, like so:

```
const BranchCount = 10;  
      WindowCount = 10;  
      InterestRate = 0.06;  
      AccountCode = 'S';
```

Any identifier that is not a reserved word can be used as a name. Any type of constant can be named in this way.

Once a constant has been given a name in a constant declaration, the identifier naming the constant can then be used anywhere that the constant is allowed, and it will have exactly the same meaning as the constant it names.

To change a named constant, you need only change the constant declaration. The meaning of all occurrences of `BranchCount` can be changed from 10 to 11 simply by changing the first 10 in the above declaration.

You can also assign a name to a quoted string with a constant declaration. For example, the following assigns the name `Name` to a long string:

*string  
constants*

```
const Name = 'Mr. E. Z. Victim';
```

---

The identifier Name can be used anywhere that the string constant can be used and will have the same meaning as the constant. In particular, if the program contains this constant declaration, then the statement

```
writeln('Program designed exclusively for ', Name)
```

will cause the following to appear on the screen:

```
Program designed exclusively for Mr. E. Z. Victim
```

Remember that you do not enclose string constant identifiers like Name in quotes. That will not produce the result you want.

Although unnamed numeric constants are allowed in a program, you should seldom use them. It often makes sense to use unnamed constants for quantities that are well known, easily recognizable, and unchangeable, such as 100 for the number of centimeters in a meter. All other numeric constants should be given names with a constant declaration, and you should use the name, rather than the unnamed constant, in the program. This will make your programs easier to read and easier to change.

## Comments

To make a program understandable, you should include explanatory notes at key places in the program. Such notes are called *comments*. In Pascal and most other programming languages, there are provisions for including such comments within the text of a program.

In Pascal a comment may be inserted almost anywhere, as long as it is preceded by the symbol {, sometimes called a “brace” or “curly bracket,” and is followed by the matching symbol }. The compiler simply ignores anything between a matching pair of curly bracket symbols { }. Comments cannot appear inside a quoted string. Otherwise, there would be no way to include the symbols '{' and '}' inside a quoted string. Further, comments cannot appear inside other comments; the effect of a comment inside of a comment is unpredictable. Except for these restrictions, a comment can go anywhere that the blank symbol is allowed. It may extend across more than one line, as long as it begins with the curly bracket symbol { and ends with the matching symbol }. Pascal does not demand a pair of the symbols { } on each line of a comment.

If the symbols { } are not available on your keyboard, or if you do not want to use them for some reason, you can use the pair (\* and \*) instead.

when :o  
comme.it

Each program unit of any substantial size or complexity should be explained by a comment. In particular, each program should open with a comment that explains what the program does, as in the following sample heading:

```
program PropertyCost(input, output);
{Accepts property assessed value, property tax rate, mortgage rate, and
loan balance as input. Computes the annual after-tax cost of the property.
Assumes full depreciation write-off as a business expense.
Uses 30-year straight line depreciation.}
```

In this book, comments will always be written in italic typeface in order to make them stand out from the program text.

It is difficult to say just how many comments a program should contain. The only correct answer is “just enough,” and this answer does not convey a lot to the novice programmer. It will take some experience to get a feel for how and when it is best to generate comments. Whenever something is important and not obvious, it merits a comment. However, too many comments are as bad as too few. A program that has a comment on each line can be so buried in comments that it hides the structure of the program and obscures the critical comments in a sea of obvious observations. Comments like the following contribute nothing to understanding and should not appear in a program:

```
Distance := Speed * Time; {Computes the distance traveled}
```

## Pitfall

### Forgetting a Closing Comment Delimiter

If you omit a closing comment delimiter '}', you might expect the compiler to notice this and issue an error message. However, this is not necessarily the case. In many cases, this error may simply cause a portion of the program to become part of one large inadvertent comment. By way of example, consider the following piece of code:

```
{Next adjust pay
Pay := Pay + Bonus;
{Tax includes state and federal tax.}
Pay := Pay - Tax;
```

Because the ending comment delimiter has been omitted from the first line displayed, the compiler will pair the opening delimiter '{' with the next closing delimiter converting the poor employee's bonus into a mere comment. The compiler will give no error message and will appear to run normally, but the following statement will never be executed because it is now inside a comment:

```
Pay := Pay + Bonus
```

What about the extra '{', you might ask. Does it not produce an error message to warn you? Probably not. Most Pascal compilers pair a '{' with the first matching '}' and happily ignore any intervening occurrence of '{', which might hint at a mistake.

## Formatted Output

When using `write` or `writeln`, you can specify the desired number of spaces used to display each value that is output. To do so, simply add a colon and a number after the expression, variable, or quoted string to be output. The number following the colon is



*field  
widths*

called a *field width*, and it specifies the total number of spaces allocated for outputting the number or other type of value. For example,

```
N := 123;
writeln('Start-field', N:5, 'End-field');
```

produces the following output (there are two spaces between the d and the 1:)

```
Start-field  123End-field
```

Any extra spaces are always in front of the value being output. If you allow too few spaces, it is not a disaster; the computer will allocate more space, but the format may not be what you desired.

*output as  
a subtask*

In your first few programs it may be best to omit field width specifications and settle for the spacing that the system decides on. It makes sense to omit such detail on your first version of a program. First, get your program to work. Then go back and add the field widths if you want neater output. This is a good example of the divide-and-conquer strategy. The task of designing a program can frequently be subdivided into two main subtasks: computing some quantities, and displaying them in a neat and clear manner. When solving the problem of how to compute the quantities, there is no need to confuse the issue with questions about the number of spaces needed to output the quantities. That is a separate task.

In this book we will frequently omit field width specifications from the `write` and `writeln` statements in our sample programs. In many cases, the program presented is only a solution to the task of computing quantities. In order to get a program with neat-looking output, it may be necessary to add some field width specifications.

*field  
specifications  
for reals*

At least one case definitely demands a field width and other related output *field specifications* in order to avoid looking ridiculous. When the output is an amount of dollars and cents, an output without a field specification usually looks absurd.

```
Total cost including tax is $ 1.563476900000000E01
```

The above screen display is a poor way to say that the cost is \$15.63. Adding the field width specification `:6:2` will convert the output into a reasonable format. The first number, the 6, says to allow a total of six spaces for the output. The second number, the 2, says to allow two digits after the decimal point. This second number is also preceded by a colon. If the second number is present, it tells the computer not to use the E notation. Hence, with the width specification `:6:2` the six spaces are allocated as follows: one for the decimal point, two after the decimal point, and three in front of the decimal point. As an example, consider the following two lines from a Pascal program:

```
COST := 15.63;
writeln('Total cost including tax is $', COST:6:2)
```

These two lines produce the following output (there is one space between the \$ and the 1):

```
Total cost including tax is $15.63
```

## Example Using Named Constants and Formatted Output

The program in Figure 3.1 was designed for a new, not yet federally or state chartered, savings institution. This institution is installing an automated 24-hour teller and has a limited amount of capital to spend on hardware and software. The program in Figure 3.1 was designed to handle deposits.

### Program

```

program Teller(input, output);
{Accepts deposit amounts as input and writes out the value
of the deposit plus interest after one year.}
const Name = 'FLY BY NIGHT THRIFT';
      Motto = 'We'll take your money any time.';
      BrCount = 2;
      CountWidth = 2;
      Rate = 7.25;
      RateWidth = 5; {Field width for outputting the Rate.}
      MoneyWidth = 7; {Field width for outputting the
                       Deposit and the Deposit plus interest.}
var Deposit, Interest, Amount: real;

begin
  writeln('WELCOME TO ', Name);
  writeln(Motto);
  writeln('We currently pay:');
  writeln(Rate : RateWidth:2, '% on deposits,');
  writeln('AND have');
  writeln(BrCount : CountWidth, ' offices to serve YOU!');
  writeln;
  writeln('ENTER the amount of your DEPOSIT at');
  writeln('the keyboard and press the RETURN KEY. ');
  writeln('PLEASE, do NOT type in a $ sign. ');
  readln(Deposit);
  writeln('Next, put your money in an ENVELOPE, ');
  writeln('WRITE your NAME on the envelope, ');
  writeln('and slip it UNDER THE DOOR. ');
  writeln('Thank you for your deposit of: ');
  writeln('$', Deposit : MoneyWidth:2);

  Interest := (Rate/100) * Deposit;
  {The division by 100 changes the percent figure to a fraction.}
  Amount := Deposit + Interest;

```

**Figure 3.1**  
Comments and  
named constants.

```
writeln('In just one short year');
writeln('your deposit will grow to');
writeln('$', Amount :MoneyWidth:2);
writeln('Thank you for choosing ', Name, '!')
end.
```

### Sample Dialogue

WELCOME TO FLY BY NIGHT THRIFT  
 We'll take your money any time.  
 We currently pay:  
     7.25% on deposits,  
 AND have  
     2 offices to serve YOU!

ENTER the amount of your DEPOSIT at  
 the keyboard and press the RETURN KEY.  
 PLEASE, do NOT type in a \$ sign.

**100.00**

Next, put your money in an ENVELOPE,  
 WRITE your NAME on the envelope,  
 and slip it UNDER THE DOOR.

Thank you for your deposit of:

\$ 100.00

In just one short year  
 your deposit will grow to

\$ 107.25

Thank you for choosing FLY BY NIGHT THRIFT!

**Figure 3.1**  
 (continued)

Since the board of directors is not sure that their current name represents sound marketing practice, the name has been placed in a constant declaration. This makes it easier to change the name if they later decide that they prefer something more dignified, such as Nocturnal Aviators Savings and Thrift Association, Inc. The motto has also been placed in a constant declaration, as have almost all other constants in the program. Even the field widths for formatted output are given names. For example, the fourth `writeln` is equivalent to

```
writeln(Rate:5:2, '% on deposits,')
```

---

## Allowable Range for Numbers

For each implementation of Pascal, there is a largest allowable positive number of type `integer` and a smallest allowable negative number of type `integer`. The language Pascal has a predefined constant called `maxint` that is equal to the largest value of type `integer` that can be used on the computer. You do not need to include it in a

*maxint*



constant declaration. It is already defined for you. The smallest value of type `integer` is not necessarily `minus maxint`, but it will be close to that value. To discover the largest possible integer value for your machine, simply embed the following in a complete program:

```
writeln('Largest integer = ', maxint)
```

Like the numbers of type `integer`, there is also a largest positive and a smallest negative number of type `real` that the computer can handle, and these numbers will vary from one installation to another. There are no predefined constants for these values, and so it is not as easy to discover these limits. However, the largest allowable number of type `real` is always much larger than the largest allowable number of type `integer`, and the smallest allowable negative number of type `real` is always much smaller than the smallest negative number of type `integer`.

---

## More about Commenting

One way to comment a program is to insert comment statements stating what the programmer expects to be true when the program execution reaches that statement. Such comments are often called *assertions* because they *assert* something that hopefully is true. For example, Figure 3.2 shows the change-making program from Figure 2.13 with comment assertions added.

```
program Change(input, output);  
{Outputs the coins used to give an amount between 1 and 99 cents.}  
var Amount, AmountLeft,  
    Quarters, Dimes, Pennies: integer;  
begin  
    writeln('Enter an amount of change');  
    writeln('from 1 to 99 cents:');  
    readln(Amount);  
  
    AmountLeft := Amount;  
  
    Quarters := AmountLeft div 25;  
    {Quarters is the maximum number of quarters in AmountLeft cents.}  
    AmountLeft := AmountLeft mod 25;  
    {AmountLeft has been decreased by the value of  
    the number of quarters specified by Quarters.}  
  
    Dimes := AmountLeft div 10;  
    {Dimes is the maximum number of dimes in AmountLeft cents.}  
    AmountLeft := AmountLeft mod 10;  
    {AmountLeft has been decreased by the value of  
    the number of dimes specified by Dimes.}
```

**Figure 3.2**  
**Program with**  
**simple comment**  
**assertions.**

**Figure 3.2**  
(continued)

```

Pennies := AmountLeft;

writeln(Amount, ' cents can be given as: ');
writeln(Quarters, ' quarters');
writeln(Dimes, ' dimes and');
writeln(Pennies, ' pennies')
end.

```

As we proceed with our study of programming techniques, we will introduce some powerful and more sophisticated ways of using assertions.

---

## Testing and Debugging

A mistake in a program is usually called a *bug*, and the process of eliminating bugs is called *debugging*. In this section we will describe the three main kinds of programming mistakes and give some hints on how to correct them.

*syntax  
errors*

The compiler will catch certain kinds of mistakes and will write out an error message when it finds one. The compiler will detect what are called *syntax errors*. The *syntax* of a language consists of the grammar and punctuation rules for the language. These rules determine whether or not your program follows the rules for the form of a Pascal program. If your program violates a syntax rule—if, for example, you have omitted a semicolon or failed to declare a variable—the compiler will issue an error statement.

*interpreting  
error  
messages*

If the compiler discovers a syntax error in your program, it will tell you where the error is likely to be and what kind of error it probably is. If the compiler says your program contains a syntax error, you can be confident that it does. However, the compiler may be incorrect about either the location of the error or its nature. It does a better job of determining the location of an error, to within a line or two, than it does of determining the source of the error. As a general rule, the compiler is likely to be right about the location of the first syntax error in your program, but it may not know what the nature of the error is. This is because the compiler is guessing at what you meant to write down and can easily guess wrong. After all, it cannot read your mind.

Error messages after the first one are more likely to be incorrect with respect to either the location or the nature of the error. Again, this is because the compiler must guess your meaning. If the compiler's first guess was incorrect, this will affect its analysis of future mistakes, since the analysis will be based on a false assumption.

Programs tend to contain numerous matching pairs, such as comment delimiters, quotes, parentheses, and other delimiters to be discussed later. A common syntax error is to miss one end of some matching pair. The compiler will always detect such an omission, but the error message it produces may be a little confusing.

As an illustration, consider the statement

```
writeln('Answers are, W, X, Y Z, 'in miles')
```

The statement has a quote missing in the first quoted string. Yet the compiler will not find a mistake until it reaches the second quoted string. The error message will probably say the error is in the neighborhood of the word 'in' and may or may not mention a missing quote. The reason for this is that the compiler perceives the quoted string

```
'Answers are, W, X, Y Z, '
```

as the first item to be written out. After all, it is a perfectly legitimate quoted string. If you realize that the mistake is in the first constant and you add the quote, you will get

```
writeln('Answers are', W, X, Y Z, 'in miles')
```

This will, of course, still produce a compiler error message pointing out the missing comma between the Y and the Z. In many cases, of which this is just one example, one mistake can hide another. In this case, the missing quote causes the compiler to ignore the missing comma.

Sooner or later you will find yourself in a situation in which you are absolutely certain that your program is correct, yet the compiler will not accept it and insists that there is a mistake in a particular line. The natural assumption is that there is a mistake in the compiler and that your program is correct. Occasionally, there are mistakes in compilers, but they are rare and it is extremely unlikely that the fault is in the compiler. Frequently it is a mistake that you cannot see for either physical or psychological reasons. You may have typed the letter "Oh" when you meant to type the digit zero. There may be a real and visible mistake that you unconsciously correct in your mind. When you cannot find anything wrong with the line, try retyping it. Amazingly enough, this will sometimes cure the problem.

There are certain kinds of errors that the computer system can detect only when a program is run. Appropriately enough, these are called *run-time errors*. Most computer systems will detect certain run-time errors and output an appropriate error message. The distinction between syntax errors and run-time errors is illustrated in Figure 3.3.

*run-time  
errors*

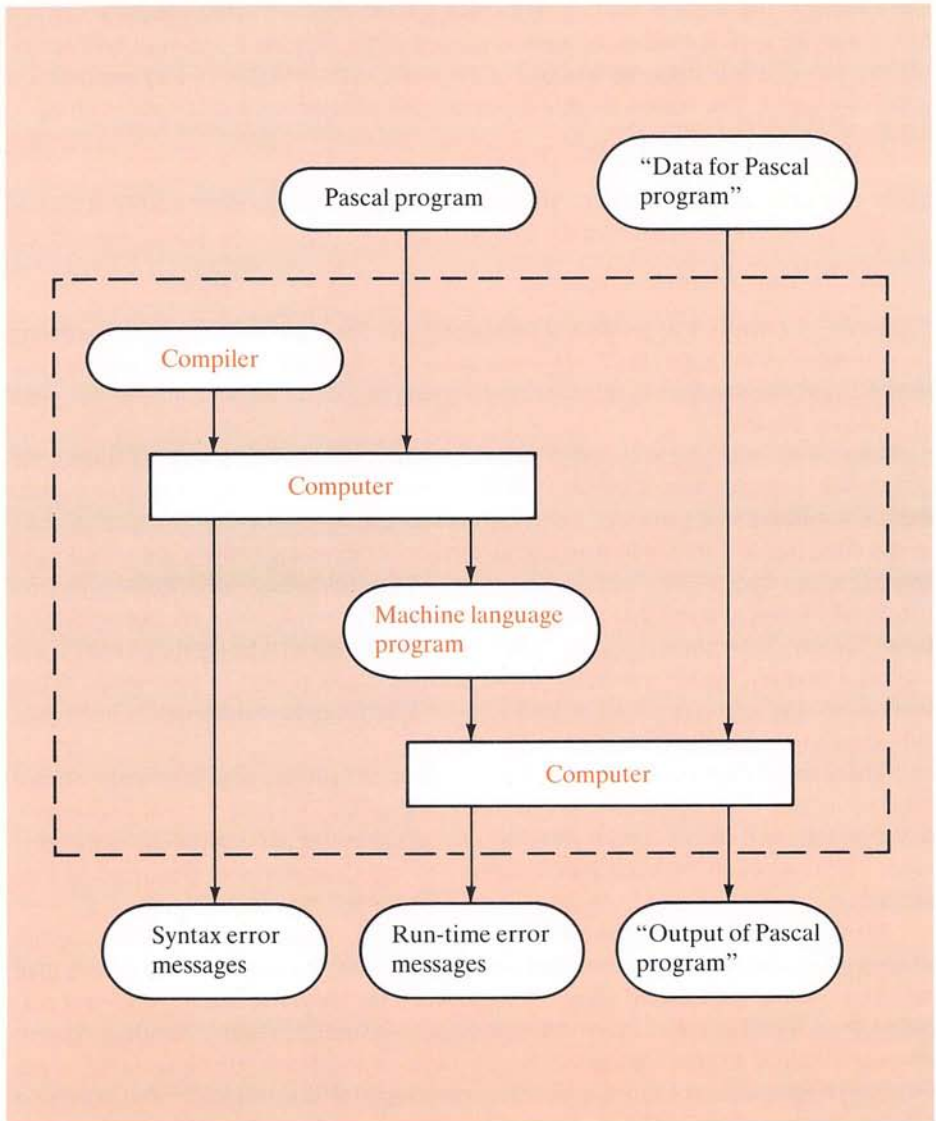
Many typical run-time errors have to do with numeric calculations. If the program attempts to evaluate an expression that would produce an integer value greater than `maxint`, the system should detect this fact when the program is run and should output an error message called an *overflow message*. A similar message should be output when real-valued expressions get to be too large or too small. (Unfortunately, some versions of Pascal do not provide overflow messages.) The system will also provide a run-time error message if the program attempts to divide by zero or to take the square root of a negative number. Other run-time errors have to do with features of Pascal that we have not yet discussed. These errors will be discussed as the relevant Pascal features are introduced.

*overflow*

If the compiler approves of your program and the program runs once with no run-time error messages, this does not guarantee that it is correct. Remember, the compiler will only tell you if you wrote a syntactically correct Pascal program. It will not tell you whether the program does what you want it to do. Mistakes in the underlying algorithm or in translating the algorithm into Pascal are called *logical errors*. If the compiler approves of your program and there are no run-time errors but the program does not

*logical  
errors*





**Figure 3.3**  
**Syntax and run-time errors.**

perform properly, then undoubtedly your program contains a logical error. Logical errors are the hardest kind to diagnose. This is because the computer gives you no error messages to help find the error. It cannot reasonably be expected to give any error messages. For all the computer knows, you may have meant what you wrote. As an example of a simple logical error, suppose that when we wrote the change-making program in Figure 3.2 we were confused about the distinction between *div* and *mod* and so we mistakenly wrote

```
AmountLeft := AmountLeft div 25
```

instead of

```
AmountLeft := AmountLeft mod 25
```

There would be no way for the computer to know that we made a mistake. As far as the computer is concerned, we did not make a mistake. We wrote a completely legitimate program. Unfortunately, it was not exactly the program we wanted. If you insert this logical error into the program in Figure 3.2, it will compile and run without precipitating an error message. It will give output in the desired format. However, it will usually give an incorrect output for the numbers of dimes and pennies. This would be a logical error. There are no syntax errors and no run-time errors, but the program is incorrect nonetheless.

Other logical errors are more complicated. As another example using the same change-making program, suppose that we forgot to decrease the amount left as we gave out coins. In other words, suppose we calculated quarters, dimes, and pennies in the following incorrect way:

```
Quarters := Amount div 25;  
Dimes := Amount div 10;  
Pennies := Amount div 1
```

This also would be a logical error. The program would still run, but it would give incorrect output.

These sample errors may seem ridiculously naive. It may seem that they are unlikely to occur and, moreover, that they would be easy to find if you did make one of them. This is not so at all. Remember, all errors are obvious once they are discovered. But at the time that you make an error it is always an undiscovered, hidden error. Otherwise, why would you make it?

A *listing* of a program is a copy of the program printed on paper and is normally produced by a printer. When you are debugging a program, it helps to have a listing. This gives you a view of the entire program and also makes it easy to write notes on the program.

*make a  
listing*

In order to test a new program for logical errors, you should run the program on several sets of representative data and check the program's performance on those inputs. If the program passes those tests, you can have more confidence in it, but you still cannot be absolutely sure that the program is correct. It still may not do what you want it to do when it is run on other data.

*testing*

The only way to justify confidence in a program is to program carefully and so avoid most errors. This approach is far better than trying to fix a program that is riddled with errors. The errors may go undetected, and even if you do detect the presence of an error, it may not be easy to locate its source.

---

## Tracing

Sometimes simply looking at the output of a program does not provide enough information to locate a logical or other type of error. If the program gives incorrect output, you know it is wrong, but you may not know where the mistake is. One way to find out

more about a program is to write out the values of variables as they change. For example, again consider the program in Figure 3.2. The value of the variable `AmountLeft` is never written out. Yet if we had mistakenly written *div* instead of *mod* in the statement

```
AmountLeft := AmountLeft mod 25
```

then the quickest way to notice this would be to notice that the value of `AmountLeft` is incorrect. To help locate mistakes such as this when testing the program, it is a good idea to insert temporary output statements for variables that change but that are not otherwise output. For example, in the program in Figure 3.2 we might insert

```
writeln('AmountLeft = ', AmountLeft)
```

This is called *tracing*.

Trace statements are temporary statements that will not appear in the final program. Hence, we want them to be easy to find and delete. One way to accomplish this is by labeling each one with a suitable comment, such as

```
{TEMP} writeln('AmountLeft = ', AmountLeft)
```

---

## Use of Assertions in Testing (Optional)

If your program contains assertions, you can use the assertions as a guide in deciding what variables to trace and where to place the `writeln` statements. For example, consider the following code from Figure 3.2:

```
{Quarters is the maximum number of quarters in AmountLeft cents.}  
AmountLeft := AmountLeft mod 25;  
{AmountLeft has been decreased by the value of  
the number of quarters specified by Quarters.}
```

It suggests that the value of the variable `Quarters` as well as both the old and the new values of `AmountLeft` should be output at this point. To help you interpret the output, it also helps to write out the assertion. The following would therefore be a sensible collection of temporary output statements to insert at the location of these assertions.

```
{TEMP}writeln('Quarters is the maximum number');  
{TEMP}writeln('of quarters in AmountLeft cents.');
```

```
{TEMP}writeln('Quarters= ', Quarters, 'AmountLeft= ', AmountLeft);  
AmountLeft := AmountLeft mod 25;  
{TEMP}writeln('AmountLeft has been decreased by the value of');  
{TEMP}writeln('the number of quarters specified by Quarters.');
```

```
{TEMP}writeln('AmountLeft= ', AmountLeft)
```

It is possible to place quotes around the actual assertion and then insert it in a `writeln`. However, if you do that, there is a good chance that you will leave one of the quote marks in the program when you finish your debugging and attempt to restore

---



the assertion to its former state. For this reason it is probably better to just repeat the assertion. Then you can delete entire lines rather than parts of lines when you clean up the final program. Since most editors make it easy to copy and edit lines, this need not be an onerous typing chore.

---

## Self-Test Exercises

1. What is the output produced by the following two lines (when correctly embedded in a complete program)? The variable `N` is of type `integer`.

```
N := -1234;  
writeln('START', N: 8, 'END')
```

2. What is the output produced by the following two lines (when correctly embedded in a complete program)? The variable `R` is of type `real`.

```
R := -12.345678;  
writeln('START', R: 8: 2, 'END')
```

## Interactive Exercises

3. Write a program that outputs `maxint` to the screen.
4. Type up and run the program `Teller` given in Figure 3.1. Change the name of the institution, its motto, and the interest rate, and then run it again.
5. Write a program that reads a value of type `real` and then writes it to the screen using the `writeln` statement given below:

```
writeln('START', R: 5: 2, 'END')
```

Run the program several times trying different numbers. Try some negative numbers as well as positive ones. Try a number that does not fit into the format specified, such as 1234.56.

---

What we anticipate seldom occurs;  
what we least expect generally happens.

*Benjamin Disraeli*

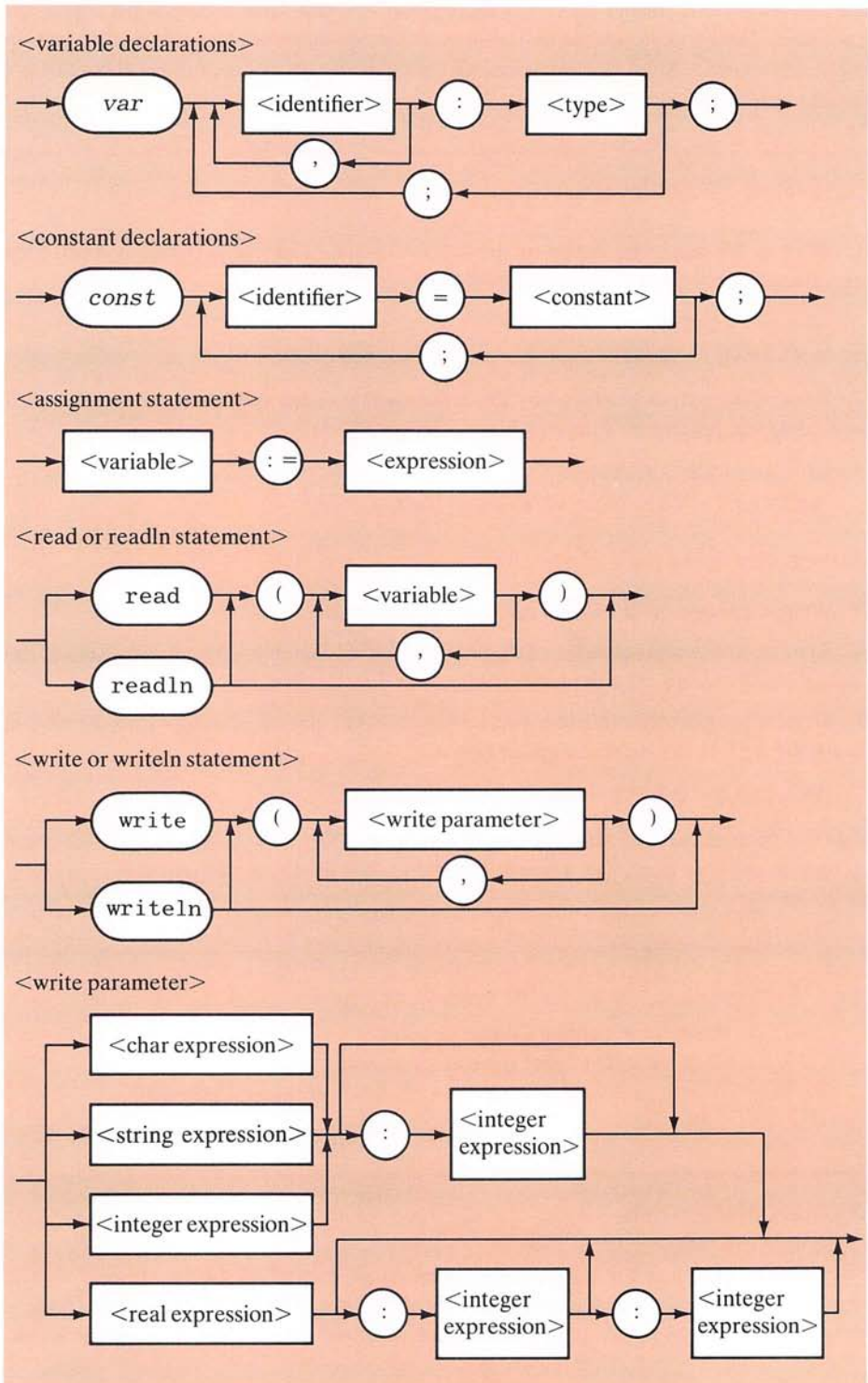
---

---

## Syntax Diagrams

The rules of grammar for a programming language are called the *syntax* rules of the language. The syntax rules determine whether or not a string of characters forms a program that the compiler can translate into machine code. There is a standard dia-

*syntax*



**Figure 3.4**  
Some syntax  
diagrams.

grammatical way to represent the syntax of programming languages. These diagrams are called, appropriately enough, *syntax diagrams*. Figure 3.4 contains syntax diagrams for many of the Pascal constructs we have presented thus far.

The procedure for using syntax diagrams is quite intuitive. Each diagram is labeled to indicate what it describes. To test whether a string of symbols satisfies the description, start at the inward pointing arrow on the left edge of the diagram and at the same time place a marker at the beginning of the questionable string; a mental marker usually works, but you can use a real marker such as your finger or a pencil point. Every time you encounter a round or rectangular box in the syntax diagram, check that your marker is at an object of the form described in the box, and then move your pointer to the next element in the string. Objects in round boxes are meant literally. For example, the box containing *var* can only match the word with the three letters v-a-r. Objects in rectangular boxes correspond to defined objects. For example, any Pascal identifier matches the box containing <identifier>. If in this way you can get completely through the syntax diagram and completely through the candidate string, then the candidate string satisfies the syntax diagram. If, for example, the diagram is labeled <constant declarations>, then the string satisfies the description of a constant declaration section.

using  
syntax  
diagrams

Since the diagrams can have branches, there is usually more than one path through a syntax diagram. A candidate string satisfies the diagram provided it matches one such path. To check the candidate, you must find the path. It is somewhat like a maze puzzle. If there is some way through the diagram, then the string passes the test. However, the diagram does not tell you how to find a path through the maze of the syntax diagram.

Syntax diagrams do not tell you about spacing, but it is intended that the objects in the boxes be separated in some appropriate way, usually by one or more blanks. Syntax diagrams give an almost complete description of the programming language syntax. Anything that fails the syntax diagram check definitely is not a correctly formed object of the kind described by the syntax diagram. A candidate that passes the syntax diagram test usually must also pass some other simple tests, such as having blanks in the right places.

We will include syntax diagrams as we explore the Pascal language. A complete set of syntax diagrams can be found in Appendix 4, but the simpler ones in the chapters will probably prove more useful.

---

## Simple Branching—If-Then-Else

Sometimes it is necessary to have a program action that chooses one of two alternatives depending on the input to the program. For example, suppose you want to design a program to compute a week's salary for an hourly employee. Assume that the firm pays an overtime rate of 1½ times the regular rate for all hours after the first 40 hours worked. As long as the employee works 40 or more hours, the pay is then equal to

$$\text{Rate} * 40 + 1.5 * \text{Rate} * (\text{Hours} - 40)$$


---



**Program**

```

program Payroll(input, output);
{Computes weekly pay for an hourly employee.}
const MaxRegTime = 40; {Max number of hours at the regular rate.}
      OTFactor = 1.5; {Factor for overtime.}
var Hours: integer;
    Rate, GrossPay: real;
begin
  writeln('Enter the hourly rate of pay. ');
  readln(Rate);
  writeln('Enter the number of hours worked, ');
  writeln('rounded up to a whole number of hours. ');
  readln(Hours);

  if Hours > MaxRegTime then
    GrossPay := Rate * MaxRegTime +
      OTFactor * Rate * (Hours - MaxRegTime)
  else
    GrossPay := Rate * Hours;

  writeln('Hours = ', Hours);
  writeln('Hourly pay rate = $', Rate);
  writeln('Gross pay = $', GrossPay)
end.

```

**Sample Dialogue 1**

```

Enter the hourly rate of pay.
20.00
Enter the number of hours worked,
rounded up to a whole number of hours.
30
Hours =      30
Hourly pay rate = $    20.0000
Gross pay = $    600.0000

```

**Sample Dialogue 2**

```

Enter the hourly rate of pay.
10.00
Enter the number of hours worked,
rounded up to a whole number of hours.
41
Hours =      41
Hourly pay rate = $    10.0000
Gross pay = $    415.0000

```

**Figure 3.5**  
An *if-then-else* branch.

If, however, there is a possibility that the employee will work less than 40 hours, this formula will unfairly pay a negative amount of overtime. The correct pay formula for an employee who works less than 40 hours is simple:

Rate \* Hours

If both more than and less than 40 hours of work are possible, then the program will need to choose between the two formulas. In order to compute the employee's gross pay (before deductions), the program action should be

Decide whether or not Hours > 40.

If it is, execute the following assignment statement:

GrossPay := Rate \* 40 + 1.5 \* Rate \* (Hours - 40)

Otherwise (i.e., if Hours ≤ 40), execute the following:

GrossPay := Rate \* Hours

There is a Pascal statement that does exactly this kind of branching. The *if-then-else* statement chooses between two alternative actions. For example, the desired action can be accomplished with the following Pascal statement:

*if* Hours > 40 *then*

GrossPay := Rate \* 40 + 1.5 \* Rate \* (Hours - 40)

*else*

GrossPay := Rate \* Hours

A complete program that uses this statement is given in Figure 3.5. In the program we have given names to the constants 40 and 1.5. If the computation had been more complicated, it would have been a good idea to use named constants even in the pseudocode.

The form of an *if-then-else* statement is as follows:

*if* <expression> <comparison operator> <another expression> *then*

<first Pascal statement>

*else*

<second Pascal statement>

The two Pascal statements may be any Pascal statements. When the program reaches the *if-then-else*, exactly one of the two embedded statements is executed. The two expressions are evaluated and compared. If the comparison is true, then the first statement is performed. If the comparison fails, then the second statement is executed.

Any two arithmetic expressions may be compared using the following comparison operators: *greater than*, *less than*, *greater than or equal to*, *less than or equal to*, *equal to*, and *not equal to*. A list of these *relational operators* is given in Figure 3.6. Some of the operators are formed using two symbols because most keyboards do not have symbols such as ≠ and ≤. For example, the Pascal expression to use in place of ≠ is the symbol pair <>. These symbol pairs are considered to be single items; do not insert any spaces between the two symbols.

These comparisons are a special case of a more general class of expressions called “boolean expressions.” We will discuss the general case in Chapter 6. For now, we will

*relational  
operators*

Math	Pascal	English	Pascal Sample	Math Equivalent
=	=	equal to	Ans = 'N'	Ans = 'N'
≠	<>	not equal to	X <> Y	X ≠ Y
<	<	less than	X < 2	X < 2
≤	<=	less than or equal to	X <= 1	X ≤ 1
>	>	greater than	Y > 0	Y > 0
≥	>=	greater than or equal to	Y >= 1	Y ≥ 1

**Figure 3.6**  
List of relational  
operators.

only use arithmetic expressions and operators of the forms just described, plus some simple comparisons of character values.

You can use an *if-then-else* statement to compare two character values to see whether they are equal or unequal. For example, suppose that *Ans* is a variable of type *char* and the program is about to execute the following *if-then-else* statement:

```
if Ans = 'Y' then
    writeln('I guess you said Yes.')
else
    writeln('You did not say Yes.')
```

If the value of *Ans* is 'Y', the first *writeln* will be executed. If its value is anything other than 'Y', the second *writeln* statement will be executed.

*ordering of  
characters*

When relational operators such as < and <= are applied to items of type *char*, they check for alphabetic order. Hence, 'A' < 'B' is true and 'Z' < 'H' is false. However, there is no uniformity as to how Pascal treats the interaction of upper- and lowercase letters. On some systems 'a' < 'Z' evaluates to true, and on others it evaluates to false. A check for alphabetical order is guaranteed to be correct only if both letters are uppercase or both letters are lowercase.

## Pitfall

### The Equality Operator

Upper- and lowercase letters are considered to be different character values. This can sometimes lead to bewildering results when Pascal is comparing characters. For example, the comparison *Ans* = 'Y' fails if the value of *Ans* is 'y'. In Chapter 6 we will discuss ways to avoid this problem. For now, you will simply have to keep this problem in mind and program around it.

The approximate nature of *real* values can cause special problems in com-



parisons involving equality. Since values of type `real` are approximate quantities, it makes no sense to test them for *exact* equality. In particular, values of type `real` should never be used with the equality operator, since the comparison is, for all practical purposes, meaningless. This rule applies to exact inequality as well. Testing `real` values using the inequality operator `<>` is equally meaningless and equally dangerous.

---

## Optional Else

It is sometimes the case that we want one of the two alternatives in an *if-then-else* branch to do nothing at all. In Pascal this can be accomplished by omitting the *else* and its accompanying statement. For example, if the value of `Sales` is less than or equal to `Minimum` when the following statement is executed, then nothing will happen, and the program will simply proceed to the `writeln` statement.

```
if Sales > Minimum then
    Salary := Salary + Bonus;
writeln('Salary = ', Salary)
```

A program using *if-then* without the *else* part is shown in Figure 3.7. Figure 3.8 gives a syntax diagram that shows the syntax for both varieties of *if-then* statements.

### Program

```
program AgeTalk(input, output);
var Age: integer;
begin
    writeln('Please enter your age. ');
    readln(Age);
    if Age >= 18 then
        writeln('You are old enough to join the army. ');
    if Age >= 21 then
        writeln('You are old enough to drink. ');
    writeln('Good luck! ')
end.
```

### Sample Dialogue

Please enter your age.

19

You are old enough to join the army.

Good luck!

**Figure 3.7**

**Program with *if-then* statements.**

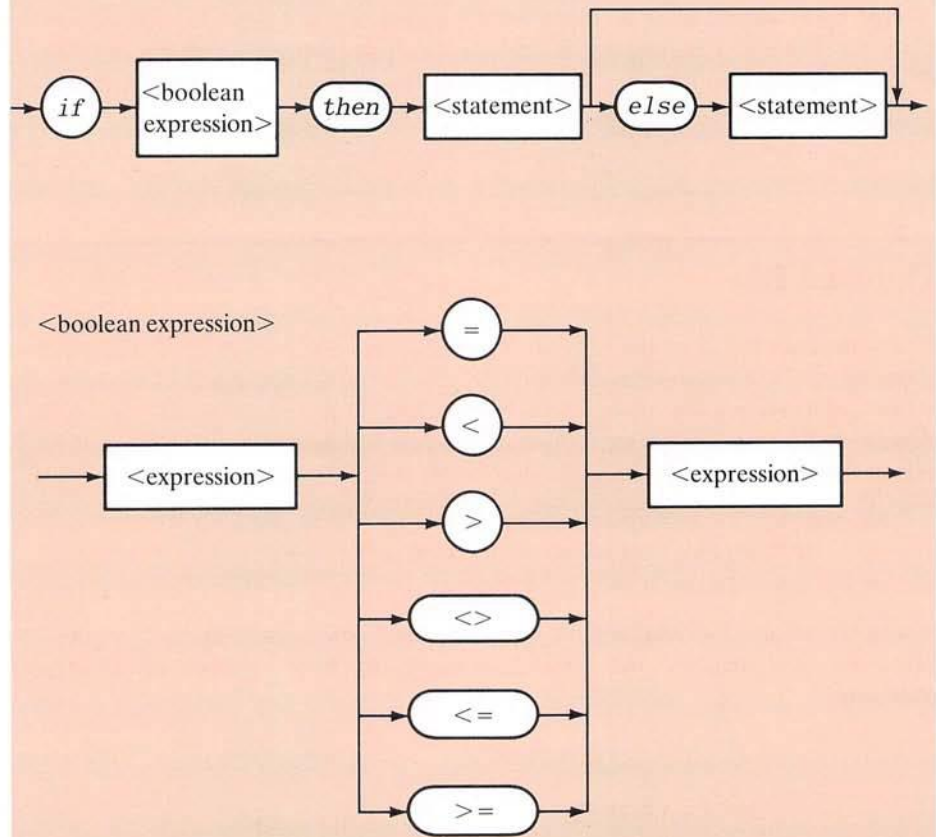


Figure 3.8  
Syntax of *if-then* and *if-then-else* statements.

```

if X > 0 then
  begin
    writeln(X, ' is positive. ');
    writeln('Wow ', X, ' is positive! ')
  end
else
  begin
    writeln(X, ' is zero or ');
    writeln('it is negative. ')
  end
end
  
```

Figure 3.9  
Compound statements used with *if-then-else*.

## Pitfalls

### Extra Semicolons

As we have stated before, semicolons are used to separate statements, but a semicolon is not a part of the statement it follows. This is of particular importance when writing *if-then-else* statements, such as

```
if N > 0 then
    writeln('Positive.')
else
    writeln('Negative or zero.');
```

```
writeln('The next statement.')
```

If you place a semicolon after the first `writeln`, your program will produce a compiler error message. The compiler will assume that the semicolon separates two statements and that the next statement starts with the identifier *else*. But that is impossible because there is no statement in Pascal that starts with *else*. If you use an extra semicolon before the *else*, the compiler will misread the statement and will produce an error message.

Now that we have statements inside of statements the terminology can get a bit complicated. Remember that the entire *if-then-else* construct is considered to be a single large statement, even though two smaller statements are embedded in it.

## Compound Statements

One often wants one or both branches of an *if-then-else* statement to execute more than one Pascal statement. To do this, enclose the statements to be executed between a *begin/end* pair, using semicolons for separators as illustrated in Figure 3.9. (It is permissible to add an extra semicolon after the last statement in the list.)

The statements, together with the *begin/end* pair, are considered to be a single Pascal statement. Statements of this form are called *compound statements*. For example, the following is a compound statement:

```
begin
    writeln(X, ' is positive.');
```

```
writeln('Wow ', X, ' is positive!')
```

```
end
```



## Iterative Enhancement

One way to design a program is to simplify the design goals so as to make the programming task easier and to then design the simpler version of the program. After that, features can be added. For example, the program for making change that we designed in the last chapter and reproduced in Figure 3.2 worked acceptably in a variety of situations. We can enhance its performance, however, by adding additional denominations of coins, such as nickels and half-dollars, to use in giving change.

This process of first designing a simple program and then adding features and refinements is sometimes referred to as *iterative enhancement*. Using this technique to develop a program makes each stage relatively easy to design and yet the final program is long, complicated, and powerful. It is not the same technique as top-down design, but it is another way to divide a large programming task into pieces that are smaller and more manageable. This approach has another very important advantage: At each stage of the process, you have a complete working program that does something meaningful and useful. This is a great psychological boost. Moreover, if you fail to achieve the final programming goal by a given deadline, you will at least have a working program and not just a collection of disconnected pieces of code.

---

## Case Study

---

### Payroll Calculation

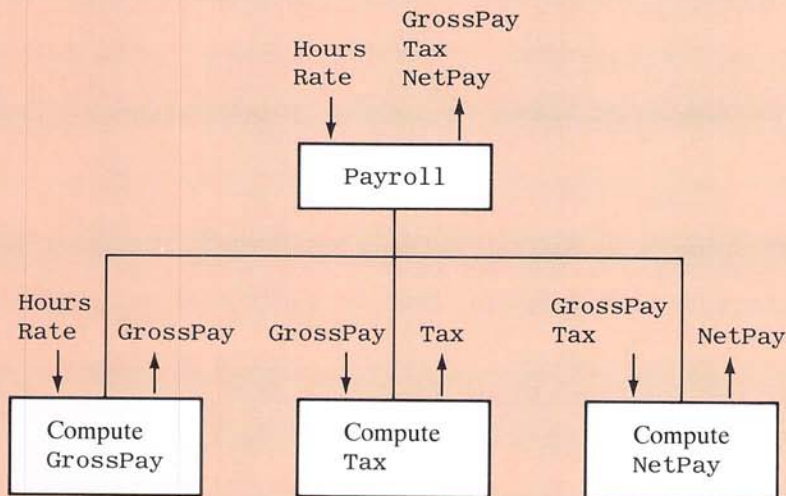
#### Problem Definition

We wish to design a program to compute the weekly pay for an hourly employee. The program will take as input the number of hours worked and the hourly pay rate and will output the gross pay as well as the net pay after taxes. To make the problem clear, we need to specify some additional details, namely how the overtime pay and tax is computed. For this problem overtime will be computed as  $1\frac{1}{2}$  times the usual rate and will apply to all hours after the first 40 hours worked. The tax will be 10% of all income over \$200.

#### Discussion

To solve this problem, we will apply the iterative enhancement technique described in the previous section. We have already produced a simple program that computes gross pay. It is the program in Figure 3.5. We will enhance it to obtain a version that also computes tax and net pay, and finally we will produce a version with formatted output.

Starting with a working program to compute gross pay, we can enhance it by adding code to compute the tax and net pay. The interaction of subtasks for the enhanced program is shown in the data flow diagram in Figure 3.10.



#### Data Summary

Hours, Rate: hours worked and hourly pay rate.

GrossPay: amount of pay before deducting tax.

Tax: tax on gross income.

NetPay: amount of pay after deduction for tax.

**Figure 3.10**  
Data flow diagram  
for complete  
payroll program.

The algorithms for computing tax and net pay follow directly from the problem definition.

```

if GrossPay > $200 then
    Tax := 10% of (GrossPay - $200)
else Tax := 0;
NetPay := GrossPay - Tax;
  
```

#### ALGORITHM

Adding this computation to the program produces the version in Figure 3.11. The final enhancement is obtained once that program is fully debugged and working, by improving the output as shown in Figure 3.12.

**Program**

```

program Payroll(input, output);
{Computes weekly pay for an hourly employee.}
const MaxRegTime = 40; {Max number of hours at the regular rate.}
      OTFactor = 1.5; {Factor for overtime.}
      Exemption = 200; {Free of tax.}
      TaxRate = 0.10; {10%}
var Hours: integer;
    Rate, GrossPay, Tax, NetPay: real;
begin
  writeln('Enter the hourly rate of pay. ');
  readln(Rate);
  writeln('Enter the number of hours worked, ');
  writeln('rounded up to a whole number of hours. ');
  readln(Hours);

  if Hours > MaxRegTime then
    GrossPay := Rate * MaxRegTime +
      OTFactor * Rate * (Hours - MaxRegTime)
  else
    GrossPay := Rate * Hours;

  if GrossPay > Exemption then
    Tax := TaxRate * (GrossPay - Exemption)
  else
    Tax := 0;

  NetPay := GrossPay - Tax;

  writeln('Hours =', Hours);
  writeln('Hourly pay rate = $', Rate);
  writeln('Gross pay = $', GrossPay);
  writeln('Tax = $', Tax, ' Net pay = $', NetPay)
end.

```

**Sample Dialogue**

```

Enter the hourly rate of pay.
20.00
Enter the number of hours worked,
rounded up to a whole number of hours.
30
Hours =      30
Hourly pay rate = $    20.0000
Gross pay = $    600.0000
Tax = $ 40.0000 Net pay = $ 560.0000

```

**Figure 3.11**  
Enhanced version  
of program in  
Figure 3.5.



**Program**

```

program Payroll(input, output);
{Computes weekly pay for an hourly employee.}
const MaxRegTime = 40; {Max number of hours at the regular rate.}
      OTFactor = 1.5; {Factor for overtime.}
      Exemption = 200; {Free of tax.}
      TaxRate = 0.10; {10%}
      Width = 7; {Total field width for money amounts.}
var Hours: integer;
    Rate, GrossPay, Tax, NetPay: real;
begin
  writeln('This program computes');
  writeln('the weekly pay for an hourly employee. ');
  writeln('All hours after the first ', MaxRegTime:4);
  writeln('are paid at a rate of');
  writeln(OTFactor:4:2, ' times the basic rate. ');

  writeln('Enter the hourly rate of pay. ');
  writeln('Do not type a dollar sign. ');
  readln(Rate);
  writeln('Enter the number of hours worked. ');
  writeln('rounded up to a whole number of hours. ');
  readln(Hours);

  if Hours > MaxRegTime then
    GrossPay := Rate * MaxRegTime +
      OTFactor * Rate * (Hours - MaxRegTime)
  else
    GrossPay := Rate * Hours;

  if GrossPay > Exemption then
    Tax := TaxRate * (GrossPay - Exemption)
  else
    Tax := 0;

  NetPay := GrossPay - Tax;

  writeln('Hours =', Hours:4);
  writeln('Hourly pay rate = $', Rate:Width:2);
  writeln('Gross pay = $', GrossPay:Width:2);
  writeln('Tax = $', Tax:Width:2,
    ' Net pay = $', NetPay:Width:2)
end.

```

**Figure 3.12**  
Further enhanced  
version of program  
in Figure 3.11.

### Sample Dialogue

This program computes  
the weekly pay for an hourly employee.  
All hours after the first 40  
are paid at a rate of  
1.50 times the basic rate.  
Enter the hourly rate of pay.  
Do not type a dollar sign.  
20.00  
Enter the number of hours worked,  
rounded up to a whole number of hours.  
30  
Hours = 30  
Hourly pay rate = \$ 20.00  
Gross pay = \$ 600.00  
Tax = \$ 40.00 Net pay = \$ 560.00

**Figure 3.12**  
(continued)

---

## Standard Functions

Pascal includes a number of *standard functions* that can appear inside arithmetic expressions. As an example, `sqrt` is the square root function. It takes a value of type either integer or real and, in either case, yields a value of type real. The value of `sqrt(4)` is 2.0, for example.

*argument*

The value that a function starts out with is called its *argument*. The value it produces is referred to as the *value returned*. In the example, 4 is the argument and 2.0 is the value returned. The argument to a function may be any expression of the appropriate type.

*value  
returned*

These standard functions can be combined with other arithmetic expressions to obtain new expressions. For example,

```
X := 2;
Y := 3 * sqrt(2 * X)
```

sets the value of Y to 6.0.

Figure 3.13 lists some commonly used standard Pascal functions as well as their descriptions and an example of each. The functions `round` and `trunc` can be used to obtain an integer value from a value of type real. The former returns the integer nearest to its argument. The latter returns the number you get by discarding the part after the decimal point. Thus, `round(4.8)` returns 5, while `trunc(4.8)` returns 4; `round(-4.8)` returns -5, while `trunc(-4.8)` returns -4. The function `sqr` computes the square of its argument. So, `sqr(X)` means the same as `X*X`. It is rather redundant, since you can use `*` to obtain the same effect. Occasionally it makes for neater notation, however, by allowing you to express the square of a long expression without having to write it twice.

---

Name	Description	Type of Argument	Type of Result	Example	Value of Example
abs	absolute value	integer	integer	abs (-2)	+2
		real	real	abs (-2.4)	+2.4
round	rounding	real	integer	round (2.6)	3
trunc	truncation	real	integer	trunc (2.6)	2
sqr	squaring	integer	integer	sqr (2)	4
		real	real	sqr (1.100)	1.2100
sqrt	square root	real or integer	real	sqrt (4)	2.00

**Figure 3.13**  
Some predefined  
Pascal functions.

The function `abs` is the absolute value function. It leaves positive numbers unchanged and removes the minus sign from negative numbers. If the argument is `real`, then `abs` returns a value of type `real`; if the argument is of type `integer`, then `abs` returns a value of type `integer`.

## Using Known Algorithms

Before you rush off to design a program from scratch, it is a good idea to see if there is a well-known algorithm to solve the task at hand. There are clever, well-known algorithms for many tasks. Often these algorithms are the product of much research and brilliant insights. In such cases, the known algorithm is likely to be better than one designed in a short time, and so you should use the known algorithm and go directly to the implementation phase.

## Case Study

### Solving Quadratic Equations

#### Problem Definition

A quadratic equation is one of the form

$$ax^2 + bx + c = 0$$

We will design a program to solve such equations for  $x$ . That is, our program will output all real number values that, when substituted for  $x$ , will make the left-hand side of the equation equal zero. The input to the program consists of the three coefficients  $a$ ,  $b$ , and  $c$ .



### Discussion

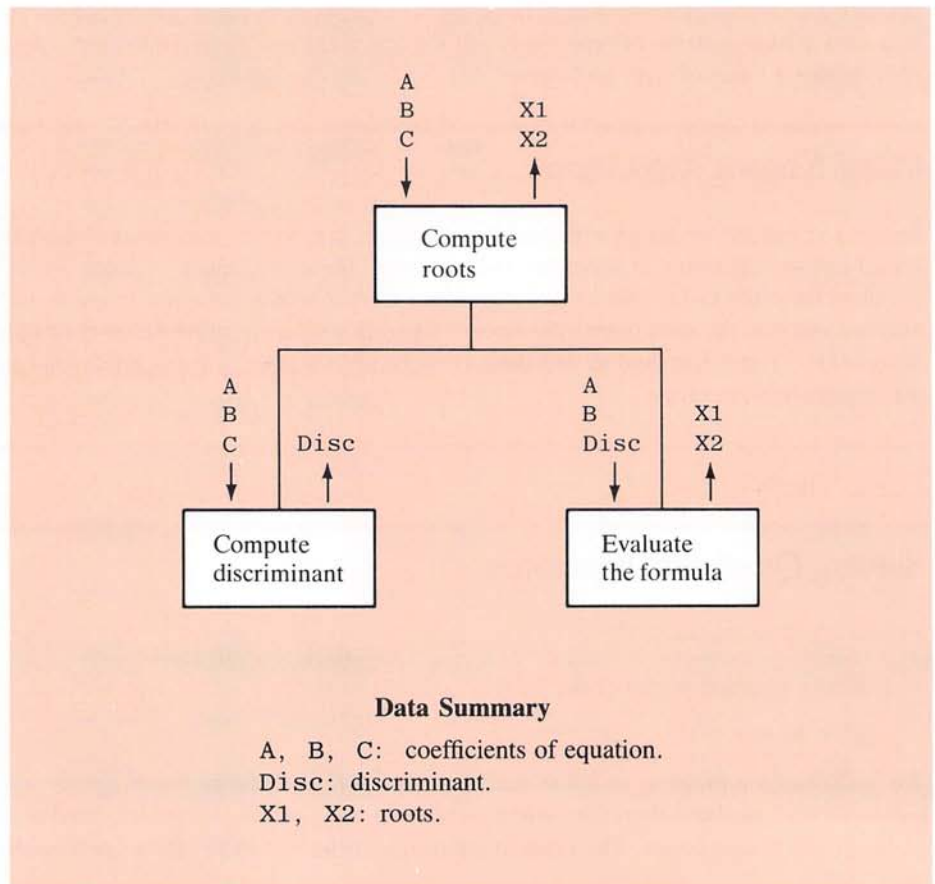
This is a well-known problem, and there is a standard formula to compute  $x$ . The equation has two roots (values of  $x$  that satisfy it). They can be computed using the two almost identical formulas

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

and

$$x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

In order for the equation to have a real number as a root, the quantity  $b^2 - 4ac$ , called the *discriminant*, must be positive. Otherwise, the formulas contain the square root of a negative number. A program to solve equations by these formulas naturally breaks down into the following subtasks:



**Figure 3.14**  
 Data flow diagram  
 for computing  
 roots of a  
 quadratic  
 equation.

1. Input coefficients.
2. Compute discriminant.
3. Evaluate the formulas.
4. Output the roots.

**ALGORITHM**

The data flow diagram is given in Figure 3.14, and the final program is given in Figure 3.15.

**Program**

```

program Roots(input, output);
{Solves quadratic equations.}
const Width = 6; {Total field width for numbers.}
      Digits = 2; {Number of digits after the decimal point.}
var A, B, C, Disc, X1, X2: real;
begin{Program}
  writeln('This program solves equations of the form:');
  writeln('a(x*x) + bx + c = 0');
  writeln('Enter the coefficients a, b, and c. ');
  readln(A, B, C);

  Disc := B * B - 4 * A * C;

  if Disc >= 0 then
    begin{then}
      X1 := (-B + sqrt(Disc)) / (2*A);
      X2 := (-B - sqrt(Disc)) / (2*A);
      writeln('For the equation ');
      writeln(A:Width:Digits, ' (x*x) + ',
        B:Width:Digits, 'x + ', C:Width:Digits, ' = 0');

      writeln('The roots are ', X1:Width:Digits,
        ' and ', X2:Width:Digits)
    end {then}
  else
    writeln('No real roots for that equation.')
  end. {Program}

```

**Sample Dialogue 1**

This program solves equations of the form:

$$a(x*x) + bx + c = 0$$

Enter the coefficients a, b, and c.

1 -2 -3

For the equation

$$1.00(x*x) + -2.00x + -3.00 = 0$$

The roots are 3.00 and -1.00

**Figure 3.15**  
Quadratic equation  
program.

### Sample Dialogue 2

This program solves equations of the form:

$a(x^2) + bx + c = 0$

Enter the coefficients a, b, and c.

2 2 2

No real roots for that equation.

**Figure 3.15**  
(continued)

*testing for  
the "impossible"*

---

## Defensive Programming

Notice that in the program in Figure 3.15 we tested to see whether the discriminant is negative, since if it is, there are no real-valued roots. If you know that a quadratic equation has a real number solution, then theoretically this need not concern you. In that case, you "know" there will not be a negative square root. However, in programming it is always wise to assume that something will go wrong. It is amazing how often it does. A good idea is to test for possible mistakes such as negative square roots or division by zero. The computer is likely to give an error statement anyway, but it may not be clear, and the error will cause the program to abort. Therefore, it is best to include a test within the Pascal program, as we have done.

---

## More about Indenting and Commenting

Indenting should display the structure of a program. If there is a statement within a statement, it is best to indicate this by indenting in some way. In particular, an *if-then-else* statement should be indented as we have been doing or in some similar manner. The important point is to use some type of indenting that shows the structure. The exact layout is not precisely dictated, but you should be consistent within any one program.

Since we now can have more than one *begin/end* pair in a program, it is a good idea to label each *begin* and *end* with a comment something like the ones in Figure 3.15. This makes it easier to find the matching *end* for each *begin*.

---

In matters of grave importance,  
style, not sincerity, is the vital thing.

*Oscar Wilde, The Importance of Being Earnest*

---



---

## Summary of Problem Solving Techniques

- Program errors can be classified into three groups: syntax errors, run-time errors, and logical errors. The computer will usually tell you about errors in the first two categories.
-



- To test for and locate logical errors, one technique is to trace variables.
- Problems can be solved and programs written in phases. First, a simple version is designed and tested and then enhancements are added in a later version.
- When designing a program, first check to see if there is a well-known algorithm for the task.

---

## Summary of Programming Techniques

- Almost all constants in a program should be given meaningful names with a constant declaration.
- Comments should be inserted to explain major subsections or any unclear part of a program. Inserting assertions is one very effective way to comment a program.
- *begin/end* pairs should be labeled to facilitate matching.
- When you use *if-then-else* statements, they should be indented to clearly display the two alternatives. Even without the *else*, some sort of indenting should be used.
- When solving problems involving numbers of type `real`, do not use a test for exact equality or exact inequality (`=` or `<>`). Since such numbers are only approximately represented in the computer, it makes no sense to test them for equality.
- It is often a good idea to test for errors such as division by zero and negative square roots within a program.

---

## Summary of Pascal Constructs

### constant declarations

Syntax:

```
const <identifier 1> = <constant 1>;  
      <identifier 2> = <constant 2>;  
      .  
      .  
      .  
      <identifier n> = <constant n>;
```

Example:

```
const Rate = 7.25;  
      Motto = 'We aim to please';  
      Days = 90;
```

The constant `<constant 1>` is given the name `<identifier 1>`, the constant `<constant 2>` is given the name `<identifier 2>`, and so forth. The identifiers can be any Pascal identifiers that are not reserved words. The identifiers can then be used anywhere that the constants can be used and will have the same value as the constants they name.

---

**maxint**

Syntax:

`maxint`

A predefined constant equal to the largest value of type `integer` that the computer system can handle.

**comments**

Syntax:

`{<text>}`

Example:

`{This is a comment}`

The text between the two symbols '`{`' and '`}`' is ignored by the compiler and so has no effect on the program. The pair '`(`' and '`)`' may be used as an alternative to '`{`'. Hence, the following is also a comment:

`(*This is also a comment*)`

The comment delimiters '`}`' and '`*)`' may not be used inside of a comment.

**formatted output**Syntax (for arguments to `write` or `writeln`):`<expression to be output> : <integer expression>`

or

`<expression to be output> : <integer expression> : <integer expression>`

Example:

`writeln(X: 6, R: 6: 2, 'Hi Mom! ': 10)`

The first number indicates the total number of spaces to be used to output the value. The value will appear at the right-hand end of that field, and any extra spaces to the left will be filled with blanks. The version with one field width number can be used with any type of output value, including quoted strings. The second integer expression is optional and can be used only for values of type `real`. The second integer expression, if present, indicates the number of digits to appear after the decimal point. The output is not in the `E` notation if a second number is specified.

**if-then-else statement**

Syntax:

```
if <expression 1> <relational operator> <expression 2> then
    <statement 1>
else
    <statement 2>
```

Example:

```
if X > 0 then
    writeln('Value of X is greater than zero')
else
    writeln('Value of X is zero or negative')
```

<statement 1> and <statement 2> may be any Pascal statements. In particular, they may be a compound statement. If the relation between the two expressions is true, then <statement 1> is executed. If the relation does not hold, then <statement 2> is executed.

### if-then statement

Syntax:

```
if <expression 1> <relational operator> <expression 2> then
    <statement>
```

Example:

```
if X > 0 then
    writeln('Value of X is greater than zero')
```

<statement> may be any Pascal statement. In particular, it may be a compound statement. If the relation between the two expressions is true, then <statement> is executed. If the relation does not hold, then no action is taken.

### compound statement

Syntax:

```
begin
    <statement 1>;
    <statement 2>;
    .
    .
    .
    <statement n>
end
```

Example (within an *if-then* statement):

```
if X > 0 then
    begin
        writeln('This is done first. ');
        writeln('This is done second. ');
        writeln('This is done last. ')
    end
```

The statements may be any Pascal statements. The effect of this statement is the same as that of executing the list of statements in order. Compound statements are often used within *if-then* and *if-then-else* statements.



## Exercises

### Self-Test Exercises

6. What is the output produced by the following code when embedded in a complete program?

```
X := 3;
if 2 > X then
    writeln('First writeln')
else
    writeln('Second writeln');
if 2 > X then
    writeln('Third writeln');
writeln('Fourth writeln')
```

7. Classify the following as true or false (when used in a Pascal *if-then-else* statement):

```
(24 mod 12) <> 0           'y' = 'Y'
'H' <= 'J'                 -12 <= maxint
```

8. Determine the value of each of the following Pascal arithmetic expressions:

```
sqrt(16)           sqrt(4 + 5)
trunc(6.8)   trunc(-6.8)   round(6.8)   round(-6.8)
abs(-6.8)    abs(6.8)      abs(4)       abs(-4)
sqrt(abs(sqr(2) - 20))
```

9. Convert the following mathematical and English expressions into Pascal boolean expressions:

$\sqrt{x} \leq y + 1$   
 Z is positive, W is not zero, X is evenly divisible by 12

10. Convert each of the following (non-Pascal) arithmetic expressions into Pascal arithmetic expressions:

$$\left(x + \frac{y}{x+z} + w\right)^2$$

$$|x - y|$$

$$\sqrt{\frac{x+3z}{w} + y}$$

### Interactive Exercises

11. Write a program that reads in an integer and outputs a statement telling whether or not the number is positive.

12. Write a program that reads in a character and outputs a statement telling whether or not the character typed in is 'n'.
13. Write a program that reads in an uppercase letter and outputs a message telling whether or not that letter follows 'M' in the alphabet.
14. Write a program that reads in an integer and outputs a statement telling whether or not the number is evenly divisible by three. If the number is not divisible by three, then the program should also output the remainder obtained when the number is divided by three. Use a compound statement inside of an *if-then-else* statement.
15. Write a program that reads in a number, computes the square root of that number using the standard function `sqrt`, squares that value using the function `sqr`, and finally outputs all three values: the input number, the square root, and the square of the square root. Run the program several times using a wide variety of numbers.
16. Write a program that reads in a decimal number, applies the functions `trunc` and `round` to the number, and then outputs both values.
17. Determine the smallest value of type `integer` that can be represented on your system. It is likely to be `-maxint` plus or minus 2.
18. Determine the smallest `real` value  $\epsilon$  such that your computer can distinguish between 1 and  $1+\epsilon$ . Feel free to use trial-and-error methods.
19. Substitute 20 for 25 in Figure 3.2 (or do something similar) and give the program to a friend to debug. Ask your friend to give you an equally perverse assignment.

## Programming Exercises

20. A liter is 0.264179 gallons. Write a program that reads in the number of liters of gasoline consumed by the user's car and the number of miles traveled by the car, and then outputs the number of miles per gallon the car delivered. Use a constant declaration.
21. The gravitational attractive force between two bodies of mass  $m_1$  and  $m_2$  separated by a distance  $d$  is given by the formula

$$F = \frac{G m_1 m_2}{d^2}$$

where  $G$  is the universal gravitational constant

$$G = 6.673 \times 10^{-8} \text{ cm}^3/\text{g sec}^2$$

Write a program that reads in the mass of two bodies and the distance between them and then outputs the gravitational force between them. The output should be in dynes; one dyne equals a  $\text{g cm/sec}^2$ . Use a constant declaration for  $G$ .

22. According to Einstein's famous equation, the amount of energy  $E$  produced when an amount of mass  $m$  is completely converted to energy is given by the formula

$$E = mc^2$$

where  $c$  is the speed of light. Write a program that reads in a mass in grams and outputs the amount of energy produced when the mass is converted to energy. The speed of light is approximately

$$2.997925 \times 10^{10} \text{ cm/sec}$$

If the mass is given in grams, then the formula yields the energy expressed in ergs. Use a constant declaration to name the speed of light.

23. Write a program to read in a weight in pounds and ounces and output the weight expressed in kilograms and grams. One pound equals 0.453592 kilograms. Use a constant declaration.

24. Write a program to read in a distance expressed in kilometers and output the distance expressed in miles. One kilometer equals 0.62137 miles. Use a constant declaration.

25. Write a program that calculates change for a cashier. The program requests the cost of the item. The user then types in the cost as a decimal numeral. The program then outputs the cost of the item including sales tax. (Use 6% as the sales tax value.) The program next requests and receives the amount tendered by the customer. Finally, the program outputs a summary of all figures, including the amount of change due to the customer. Use a constant declaration for the tax rate. Be sure to use field width specifications so as to produce reasonable-looking output.

26. Write a program to gauge the amount of inflation over the past year. The program asks for the price of an item (such as a hot dog or a one-carat diamond) both one year ago and today. It estimates the inflation rate as the difference in price divided by the price a year ago.

27. Enhance your program from the previous exercise by having it also print out the estimated price of the item one year from now and two years from now. The increase in cost in one year is estimated as the inflation rate times the current price.

28. The prices of stocks are normally given to the nearest eighth of a dollar; for example,  $29\frac{7}{8}$  or  $89\frac{1}{2}$ . Write a program that computes the value of the user's holding of one stock. The program asks for the number of shares held, the whole dollar portion of the price, and the fraction portion. The fraction portion is to be input as two integer values, one for the numerator and one for the denominator.

29. The relationship between the sides ( $a$ ,  $b$ ) of a right triangle and the hypotenuse ( $h$ ) is given by the Pythagorean formula

$$a^2 + b^2 = h^2$$

Write a program that reads in the lengths of the two sides of a right triangle and computes the hypotenuse of the triangle.

30. The area of a triangle whose sides are  $a$ ,  $b$ , and  $c$  can be computed by the formula

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where  $s = (a+b+c)/2$ . Write a program that reads in the lengths of the three sides of a triangle and outputs the area of the triangle.



31. Rewrite the program Change in Figure 3.2 so that it takes as input any amount of dollars and cents (as a number of type `real`) and outputs the correct number of bills as well as coins to give as change. Use bill denominations of \$1, \$5, and \$20 only.
32. An hourly employee is paid at a rate of \$9.73 per hour for up to 40 hours worked per week. Any hours over that are paid at the overtime rate of  $1\frac{1}{2}$  times that. From the worker's gross pay, 6% is withheld for social security tax, 14% is withheld for federal income tax, 5% is withheld for state income tax, and \$6 per week is withheld for union dues. If the worker has three or more covered dependents, an additional \$10 is withheld to cover the extra cost of health insurance beyond that paid by the employer. Write a program that takes as input the number of hours worked in a week and the number of dependents and then outputs the worker's gross pay, each withholding, and the net take-home pay for the week.
33. Write a program that computes the cost of postage on a first-class letter according to the following rate scale: 24 cents for the first ounce or fraction of an ounce, 8 cents for each additional half ounce, plus a \$5 service charge if the customer desires special delivery.

---

## References for Further Reading

The following are reference manuals rather than tutorial books. The first two are especially designed for TURBO Pascal. The third describes the differences between various Pascal implementations. It is particularly useful if you are changing from one Pascal system to another, different Pascal system.

*TURBO Pascal 4.0, Owner's Handbook*, 1987, Borland International, Inc., Scotts Valley, Ca.

*TURBO Pascal User's Guide Version 5.0*, 1988, Borland International, Inc., Scotts Valley, Ca.

J. Tiberghien, *The Pascal Handbook*, 1981, Sybex, Berkeley, Ca.

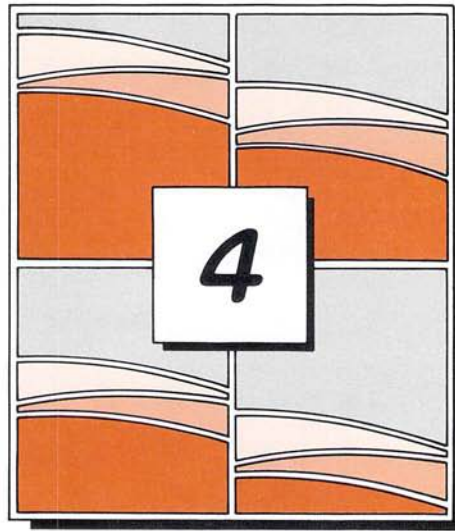
The following reference manuals are designed for individuals who already know another programming language besides Pascal:

K. Jensen and N. Wirth, *Pascal Users Manual and Report*, 1974, Springer-Verlag, New York.

D.L. Matuszek, *Quick Pascal*, 1982, John Wiley, New York.

---





## ***Designing Procedures For Subtasks***

Little strokes fell great oaks.

*Benjamin Franklin,  
Poor Richard's Almanac*



## Chapter Contents

Simple Pascal Procedures  
Variable Parameters  
Parameter Lists  
Implementation of Variable  
Parameters (Optional)  
Procedures Calling Procedures  
Procedural Abstraction  
Self-Test Exercises  
Value Parameters  
What Kind of Parameter to Use  
Mixed Parameter Lists

Case Study—Supermarket Pricing  
Case Study—Change Program with  
Procedures  
Pitfall—Incorrectly Ordered  
Parameter Lists  
Generalizing Procedures  
Choosing Parameter Names  
Summary of Problem Solving and  
Programming Techniques  
Summary of Pascal Constructs  
Exercises

A good way to design an algorithm to solve a task is to break it down into subtasks and solve these subtasks by smaller, simpler algorithms. Ultimately, the subalgorithms to solve these subtasks are translated into Pascal code, and the entire larger algorithm containing these subalgorithms is translated into a Pascal program. Since the subalgorithms are algorithms, it is natural to think of them as smaller programs within a larger program. Moreover, preserving this structure in the final Pascal program makes the program easier to understand, easier to change if need be and, as will become apparent, easier to write, test, and debug. Pascal, like most programming languages, has facilities to include programlike entities inside of programs. The Pascal term for such programlike entities is *procedure*. In this chapter we introduce procedures and give some guidelines for using them effectively.

---

## Simple Pascal Procedures

In Pascal you can assign a name to a sequence of statements by means of something called a *procedure declaration*. Using the procedure name inside the body of the program will then have the same effect as executing the sequence of statements. For example, the following is a sample procedure declaration that assigns the name `Compliment` to two `writeln` statements:

```
procedure Compliment;  
begin{Compliment}  
  writeln('A lovely letter. ');  
  writeln('One of my favorites. ');  
end; {Compliment}
```

In a program containing this procedure declaration, the identifier `Compliment` can be used anywhere a statement can be used, and when it is executed, it will cause the following to appear on the screen:

```
A lovely letter.  
One of my favorites.
```

This is illustrated in Figure 4.1.

The syntax for a simple procedure declaration is as follows. The first part of the procedure is called the *heading*. It consists of the word *procedure* followed by a (non-reserved-word) identifier to serve as the procedure name, followed by a semicolon. This is followed by the *body* of the procedure. The body of the procedure is a list of statements separated by semicolons and enclosed between a *begin* / *end* pair; in other words, the body of a procedure is a compound statement. You end a procedure declaration with a semicolon placed after the final *end*. In a Pascal program, procedure declarations are placed after the variable declarations and before the main body of the program. The order of declarations within a program is summarized in Figure 4.2.

A procedure name occurring inside the body of a program is considered to be a special kind of statement known as a *procedure call* or a *procedure invocation*. These procedure-call statements are treated just like any other kind of statement when it comes to syntactic details such as the placement of semicolons.

---

## Variable Parameters

Suppose we want to design a program to help a cashier total the cash on hand at the end of a work day. The cashier counts and enters the number of each coin; the program then computes the dollar value of the coins. To avoid input errors, each input statement is prefaced by a warning of exactly what to enter and how to enter it. The program might start out as follows:

```
writeln('Enter the number of half-dollar coins. ');  
writeln('Do not total the amount. ');  
writeln('Just enter the number of coins. ');  
readln(HalfDollars)
```

---

*procedure  
declaration*

*use of  
procedures*

*procedure  
declaration  
syntax*

*procedure  
call*

The program might next request the number of quarters, as follows:

```
writeln('Enter the number of quarters. ');
writeln('Do not total the amount. ');
writeln('Just enter the number of coins. ')
readln(Quarters)
```

### Program

```
program BriefEncounter(input, output);
var FirstI, LastI: char;

procedure Compliment;
begin{Compliment}
  writeln('A lovely letter. ');
  writeln('One of my favorites. ')
end; {Compliment}

begin{Program}
  writeln('Please enter your first initial. ');
  readln(FirstI);
  Compliment;
  writeln('Now enter your last initial. ');
  readln(LastI);
  Compliment;
  writeln('Pleased to meet you ', FirstI, '.', LastI, '. ')
end. {Program}
```

### Sample Dialogue

```
Please enter your first initial.
J
A lovely letter.
One of my favorites.
Now enter your last initial.
R
A lovely letter.
One of my favorites.
Pleased to meet you J.R.
```

**Figure 4.1**  
Program with a  
procedure.

**Figure 4.2**  
Order of  
declarations.

- 
1. Constant declarations.
  2. Variable declarations.
  3. Procedure declarations.
-



Notice that the last three lines of these two pieces of code perform the same task, namely prompting the user with input instructions and then reading one number. Since this is a repeated subtask, it makes sense to declare these three lines as a procedure. There is, however, one problem with this idea. The last line is slightly different in these two pieces of code. One time we use the variable `HalfDollars`, and one time we use the variable `Quarters`. One solution would be to make the procedure only two lines long and omit the last of the three lines from the procedure, just as we are choosing to omit the first of these lines. That will work, but it is not a very good solution. What we really want is a procedure that has a blank that can be filled in with the variable `HalfDollars` in the first procedure call, filled in with the variable `Quarters` in the second call, and filled in with the variables `Dimes`, `Nickels`, and `Pennies` later on. Pascal allows us to do just that.

The object which acts as a blank to be filled is called a *formal variable parameter*, and although it looks exactly like a variable, it is not a variable. It is a labeled blank that must be filled in with a variable when the procedure is called. A formal variable parameter may be any identifier other than a reserved word. A procedure called `GetNumber`, which uses a formal variable parameter called `Number` and performs the task we desire, is included in the program in Figure 4.3. To use the procedure with the particular variable `HalfDollars` substituted for the formal variable parameter `Number`, we use the following procedure call:

```
GetNumber (HalfDollars)
```

The variable in the procedure call, in this case `HalfDollars`, is called an *actual variable parameter*. When the procedure is called, every occurrence of the formal variable parameter within the body of the procedure declaration is replaced by the actual variable parameter, and then the statements in the procedure body are executed. This process is sometimes called *parameter passing*. As the program in Figure 4.3 illustrates, a procedure may be called more than once, using different actual parameters each time.

Notice that a formal variable parameter has a type that must be stated in the procedure heading and that the formal and actual parameters must agree in type. The type specification for a formal variable parameter is given in parentheses after the procedure name and looks very much like a variable declaration. Since the type of the parameter is given in the procedure heading, it need not be given anywhere else. In particular, it should not be declared in the variable declarations of the program. After all, it is not a variable. Parameters may be of any type. (Hence, in `TURBO Pascal` they may be of *string* types. However, you need a bit more machinery in order to use *string* parameters, and so we will postpone their introduction until Chapter 8.)

*formal  
variable  
parameters*

*actual  
variable  
parameters*

---

## Parameter Lists

A procedure can have any number of formal variable parameters. They are simply all listed in parentheses after the procedure name in the procedure heading and are separated by semicolons. For example, one procedure heading might be

```
procedure Total (var P1: integer;  
                 var P2: real; var P3: real);
```

*formal  
parameter  
list*

---

**Program**

```

program TotalChange(input, output);
  {Reads in the count of each type of coin and outputs their total value.}
  const MoneyLength = 7; {Field length for total amount.}
  var HalfDollars, Quarters, Dimes,
      Nickels, Pennies: integer;
      Total: real;

  procedure GetNumber(var Number: integer);
    {Writes instructions to the user, and then reads a number of
     coins from the keyboard and stores that number in Number.}
    begin{GetNumber}
      writeln('Do not total the amount. ');
      writeln('Just enter the number of coins. ');
      readln(Number)
    end; {GetNumber}

  begin{Program}
    writeln('Enter the number of half-dollar coins. ');
    GetNumber(HalfDollars);

    writeln('Enter the number of quarters. ');
    GetNumber(Quarters);

    writeln('Enter the number of dimes. ');
    GetNumber(Dimes);

    writeln('Enter the number of nickels. ');
    GetNumber(Nickels);

    writeln('Enter the number of pennies. ');
    GetNumber(Pennies);

    Total := 0.50*HalfDollars +
             0.25*Quarters +
             0.10*Dimes +
             0.05*Nickels +
             0.01*Pennies;

    writeln(HalfDollars, ' half-dollars, ',
            Quarters, ' quarters, ');
    writeln(Dimes, ' dimes, ', Nickels, ' nickels and ');
    writeln(Pennies, ' pennies ');
    writeln('total to: $', Total :MoneyLength:2)
  end. {Program}

```

**Figure 4.3**

**Program that uses  
parameter passing.**

### Sample Dialogue

```

Enter the number of half-dollar coins.
Do not total the amount.
Just enter the number of coins.
12
Enter the number of quarters.
Do not total the amount.
Just enter the number of coins.
325
Enter the number of dimes.
Do not total the amount.
Just enter the number of coins.
103
Enter the number of nickels.
Do not total the amount.
Just enter the number of coins.
107
Enter the number of pennies.
Do not total the amount.
Just enter the number of coins.
57
    12 half-dollars, 325 quarters,
    103 dimes, 107 nickels and
    57 pennies
total to: $ 103.47

```

**Figure 4.3**  
(continued)

When two or more formal variable parameters are of the same type and occur one after the other in the list of parameters, they may be combined so that their type need only be written once. In that case, the combined parameters are separated by commas. For example, the following procedure heading is equivalent to the one just given:

```
procedure Total(var P1: integer; var P2, P3: real);
```

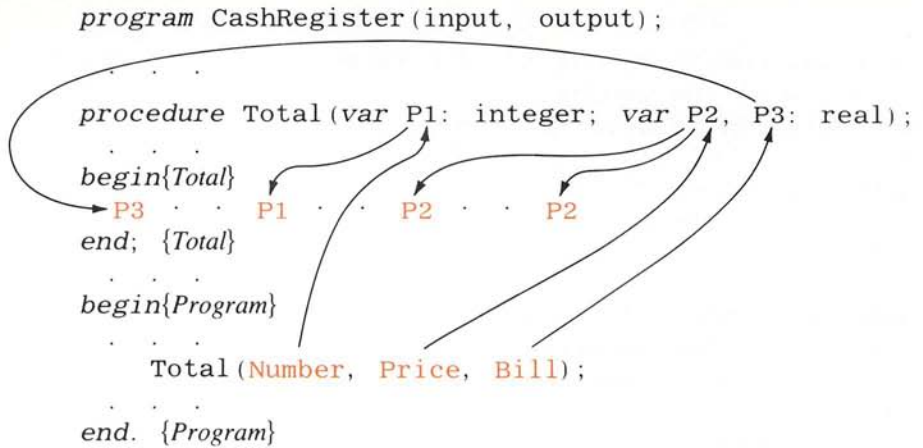
Notice that there is a *var* for each group of formal variable parameters.

When the procedure is called, the actual variable parameters are given in parentheses after the procedure name. When the procedure call is executed, the actual variable parameters are substituted for all occurrences of formal variable parameters inside the procedure body. The substitution follows the ordering: The first actual parameter listed in the procedure call is substituted for the first formal parameter listed in the procedure heading, the second actual parameter is substituted for the second formal parameter, and so forth. The number of actual parameters in a procedure call must always equal the number of formal parameters in the procedure declaration. Moreover, the type of each actual parameter must be the same as the type of the corresponding formal parameters it replaces. Hence, in the sample procedure call

```
Total(Number, Price, Bill)
```

*actual  
parameter  
list*





**Figure 4.4**  
**Parameter**  
**substitutions.**

the actual variable parameter *Number* must be a variable of type *integer*; the actual variable parameters *Price* and *Bill* must be variables of type *real*. *Number* is substituted for *all* occurrences of *P1* in the procedure body, *Price* is substituted for *all* occurrences of *P2*, and *Bill* is substituted for *all* occurrences of *P3*.

As indicated by this example, one minor syntactic difference between formal and actual parameters is that actual parameters are separated by commas rather than by a combination of commas and semicolons. The list of parameters, either formal in the procedure heading or actual in the procedure call, is referred to as a *parameter list*.

The order of substituting actual for formal parameters is illustrated in Figure 4.4. Figure 4.5 shows our example embedded in a complete program and illustrates one way to visualize the complete procedure-call process. First, the actual parameters are substituted for the formal parameters. Second, the body of the procedure declaration is substituted for the procedure call. Finally, the actions prescribed by the resulting code are executed. Be sure to notice that the substitution follows the ordering of the formal parameters in the formal parameter list. It does not depend on their order of occurrence inside the body of the procedure.

---

## Implementation of Variable Parameters (Optional)

Recall that program variables are typically implemented as memory locations. The compiler assigns one memory location to each variable. For example, when the program in Figure 4.5 is compiled, the variable *Price* might be assigned location 10010, *Bill* location 10011, and *Number* location 10012. For all practical purposes, these memory locations are the variables.

As we said in the previous two sections, the formal variable parameters that appear in a procedure heading are formal blanks and the actual variable parameters are vari-

---

**Program**

```

program CashRegister(input, output);
const Rate = 0.06; {Sales tax rate.}
      Width = 6; {Field width for money amounts.}
var Price, Bill: real;
      Number: integer;

procedure Total(var P1: integer; var P2, P3: real);
{Sets the value of P3 equal to the total cost of P1 items at a price of P2 each plus sales tax.}
begin{Total}
  P3 := P1 * (P2 + Rate * P2)
end; {Total}

begin{Program}
  writeln('Enter price and number of items:');
  readln(Price, Number);
  Total(Number, Price, Bill);
  writeln(Number, ' items at $', Price :Width:2);
  writeln('Total Bill $', Bill :Width:2)
end. {Program}

```

1. Substitute actual parameters for formal parameters to obtain the meaning of the procedure body:

```

begin{Total}
  Bill := Number * (Price + Rate * Price)
end; {Total}

```

2. Substitute the procedure body for the procedure call:

```

begin{Program}
  writeln('Enter price and number of items:');
  readln(Price, Number);
  begin{Total}
    Bill := Number * (Price + Rate * Price)
  end; {Total}
  writeln(Number, ' items at $', Price :Width:2);
  writeln('Total Bill $', Bill :Width:2)
end. {Program}

```

3. Execute the resulting code:

```

Enter price and number of items:
100.00 2
  2 items at $100.00
Total Bill $212.00

```

**Figure 4.5**  
**How variable**  
**parameters work.**

ables to be substituted for the blanks. For example, consider the following procedure heading from Figure 4.5:

```
procedure Total(var P1: integer; var P2, P3 :real);
```

The formal parameters P1, P2, and P3 are not variables and so have no memory locations assigned to them. They are just formal names that name nothing.

Now consider a procedure call like the following call from the same figure:

```
Total(Number, Price, Bill)
```

When the procedure call is executed, the procedure is not given the actual variable names *Number*, *Price*, and *Bill*. Instead, it is given a list of the memory locations associated with each name. In this example, the list consists of the locations

```
10012, 10010, 10011
```

which are the locations assigned to actual variable parameters *Number*, *Price*, and *Bill*, *in that order*. It is these locations that are associated with the formal parameters. The first formal parameter is associated with the first memory location, the second formal parameter is associated with the second location on the list, and so forth. Diagrammatically, in this case the correspondence is

Number	→	10012	→	P1
Price	→	10010	→	P2
Bill	→	10011	→	P3

When the procedure statements are executed, the formal parameters are interpreted as meaning the corresponding memory locations. So when P3 is changed by the assignment statement in the procedure, it is memory location 10011 that is changed. Whatever the procedure instructs the computer to do to P3 is actually done to memory location 10011, which for all practical purposes is the variable *Bill*.

Notice that the order of parameters is critically important. The procedure, when called, receives no variable names at all. It merely receives a list of memory locations, which it associates with formal variable parameters. If the actual variable parameters are not in the right order, the list of memory locations will not be in the right order, and the procedure will use the wrong memory locations and produce undesirable results.

Also notice that by using memory location addresses (numbers) instead of variable names, the compiler can easily handle otherwise confusing substitutions with coincidental collisions of names. An actual parameter may have the same name as some formal parameter, even one other than the one to which it corresponds, and the computer will not be confused.

---

## Procedures Calling Procedures

One procedure may include a call to another procedure. The only restriction is that a procedure call cannot appear before the procedure is declared. An example is given in Figure 4.6. The program in that figure is similar to the one in Figure 4.5 and in fact, the procedure *Total* is identical to the procedure of the same name in Figure 4.5. The

---



**Program**

```

program CashRegister2(input, output);

const Rate = 0.06; {Sales tax rate.}
      Width = 6; {Field width for money amounts.}
var Price, Bill: real;
    Number: integer;
    Wholesale: char; {A value of 'y' means "yes, wholesale."}

procedure Total(var P1: integer; var P2, P3: real);
{Sets the value of P3 equal to the total cost of P1 items at a price of P2 each plus sales tax.}
begin{Total}
    P3 := P1 * (P2 + Rate * P2)
end; {Total}

procedure AmountDue(var N1: integer;
                    var N2, N3: real; var C: char);
{Sets the value of N3 equal to the total cost of N1 items at a price of N2 each.
If C = 'y' no tax is added; otherwise sales tax is added.}
begin{AmountDue}
    if C = 'y' then {No sales tax.}
        N3 := N1 * N2
    else {Include sales tax.}
        Total(N1, N2, N3)
end; {AmountDue}

begin{Program}
    writeln('Enter price and number of items:');
    readln(Price, Number);
    writeln('Is this a wholesale transaction? (y/n)');
    readln(Wholesale);

    AmountDue(Number, Price, Bill, Wholesale);

    writeln(Number, ' items at $', Price :Width:2);
    writeln('Total Bill $', Bill :Width:2);
    if Wholesale = 'y' then
        writeln('no tax')
    else
        writeln('including tax')
end. {Program}

```

**Figure 4.6**  
**Procedure calling**  
**another procedure.**

1. Substitute actual parameters for formal parameters in the procedure body:

```
begin{AmountDue}
  if Wholesale = 'y' then {No sales tax.}
    Bill := Number * Price
  else {Include sales tax.}
    Total (Number, Price, Bill)
end; {AmountDue}
```

2. Substitute the procedure body for the procedure call:

```
begin{Program}
  writeln('Enter price and number of items:');
  readln(Price, Number);
  writeln('Is this a wholesale transaction? (y/n) ');
  readln(Wholesale);

  begin{AmountDue}
    if Wholesale = 'y' then {No sales tax.}
      Bill := Number * Price
    else {Include sales tax.}
      Total (Number, Price, Bill)
    end; {AmountDue}

  writeln(Number, ' items at $', Price :Width:2);
  writeln('Total Bill $', Bill :Width:2);
  if Wholesale = 'y' then
    writeln('no tax')
  else
    writeln('including tax')
  end. {Program}
```

3. Again substitute actual parameters for formal parameters to obtain the meaning of the other procedure body:

```
begin{Total}
  Bill := Number * (Price + Rate * Price)
end; {Total}
```

4. Substitute the procedure body for the remaining procedure call:

```
begin{Program}
  writeln('Enter price and number of items:');
  readln(Price, Number);
  writeln('Is this a wholesale transaction? (y/n) ');
  readln(Wholesale);
```

**Figure 4.6**  
(continued)

```

begin{AmountDue}
  if Wholesale = 'y' then {No sales tax.}
    Bill := Number * Price
  else {Include sales tax.}
    begin{Total}
      Bill := Number * (Price + Rate * Price)
    end; {Total}
end; {AmountDue}

writeln(Number, ' items at $', Price :Width:2);
writeln('Total Bill $', Bill :Width:2);
if Wholesale = 'y' then
  writeln('no tax')
else
  writeln('including tax')
end. {Program}

```

5. Execute the code:

```

Enter price and number of items:
100.00  2
Is this a wholesale transaction? (y/n)
n
    2 items at $100.00
Total Bill $212.00
including tax

```

**Figure 4.6**  
(continued)

program in Figure 4.6 allows for two possibilities: the normal retail sale, which includes sales tax, and a sale to a wholesaler, which does not include sales tax (since that will be paid when the item is finally sold retail later). The program contains a call to the procedure `AmountDue`, which directly calculates the total due for wholesale buyers, but which calls the procedure `Total` in the case of a retail sale. As shown in the figure, the procedure declaration for `Total` must precede the declaration of `AmountDue`.

Multiple levels of procedure calls are handled just like the simple one-level calls we saw earlier. With two levels of procedure calls, we simply apply the substitution mechanism twice, as shown in Figure 4.6. When the procedure call

```
AmountDue(Number, Price, Bill, Wholesale)
```

is encountered, the actual variable parameters `Number`, `Price`, `Bill`, and `Wholesale` are substituted for the formal variable parameters `N1`, `N2`, `N3`, and `C`, in that order. This substitution applies to all occurrences of the formal variable parameters, and so it applies to the procedure call

```
Total(N1, N2, N3)
```



as shown in Figure 4.6, item 1. After the substitution, this procedure call within the procedure `AmountDue` reads

```
Total (Number, Price, Bill)
```

As shown in Figure 4.6, item 2, the entire procedure body of `AmountDue`, including this call to `Total`, is then substituted into the body of the program. This still leaves one procedure call:

```
Total (Number, Price, Bill)
```

This call is handled by another application of the substitution rules. As shown in items 3 and 4 of Figure 4.6, the actual variable parameters `Number`, `Price`, and `Bill` are substituted for the formal variable parameters `P1`, `P2`, and `P3`, and the resulting code is substituted into the program body. At that point there are no procedure calls left, and the meaning of the code is clear.

The substitution mechanism we have just described correctly mirrors the computer's performance and is easy to understand, but once you become familiar with it you may prefer a variant that performs the various substitutions in another order, substituting parameters for parameters. In the example just discussed, you might first take care of the procedure call to `Total` by substituting `N1`, `N2`, and `N3` in for `P1`, `P2`, and `P3` and then substituting the body of the procedure `Total` into the body of the procedure `AmountDue`. The result is the same, although it can be a bit more difficult to keep track of.

---

## Procedural Abstraction

One function of procedures is to simplify your reasoning by *abstracting* away irrelevant properties of a program part. When using a procedure, we need only think about what the procedure does; we need not think about how it does it. When we use the procedure `Total`, which appears in Figures 4.5 and 4.6, we need only know that it computes the total bill for a purchase including sales tax; we need not concern ourselves with how it computes this amount. It happens to perform the computation with the statement

```
P3 := P1 * (P2 + Rate * P2)
```

But it could just as well have been performed with the statement

```
P3 := P1 * (1 + Rate) * P2
```

or even with the slightly less efficient

```
P3 := P1 * P2 + P1 * Rate * P2
```

This detail need not concern us when we use the procedure.

Once a procedure is written, the details of how it works can be ignored. What it does should be expressed in a comment at the start of the procedure declaration. This comment should tell anyone who wants to use the procedure all that he or she needs to know in order to use it. The user of the procedure should not need to even look at the procedure body.

---

Procedural abstraction is more than a way of summarizing a procedure's actions. It should be the first step in designing and writing a procedure. When you design a program, you should specify what each procedure does before you start to design how the procedure will do it. In particular, the comment that describes a procedure's actions and the list of parameters that shows which items may be affected by the procedure should be designed and written down before you start to design the procedure body. If you later discover that your specification cannot be realized in a reasonable way, you may need to back up and rethink what a procedure should do, but by clearly specifying what you think the procedure should do, you will minimize both design errors and time wasted writing code that does not fit the task at hand.

Procedural abstraction is a way of clarifying your reasoning about your personal programs. It is even more important when programming as a team. In team situations, one programmer often does not know how a procedure written by another programmer works, and need not know. In fact, you have already experienced this. You have used the predefined function `sqrt`, which calculates square roots. You can use this function effectively even if you know absolutely no algorithm for extracting square roots, and you certainly do not need to know the details of how your particular implementation extracts square roots.

---

This one thing I do, forgetting those things which are behind,  
and reaching forth unto those things which are before,  
I press toward the mark.

*The Epistle of Paul the Apostle  
to the Philippians 3:13–14*

---

## Self-Test Exercises

1. What is the output of the following program?

```
program Exercise1(input, output);
procedure Friendly;
begin{Friendly}
  writeln('Hello')
end; {Friendly}
procedure Shy;
begin{Shy}
  writeln('Goodbye')
end; {Shy}
begin{Program}
  writeln('Begin Conversation');
  Shy; Friendly;
  writeln('One more time:');
  Friendly; Shy;
  writeln('End Conversation')
end. {Program}
```

2. What is the output of the following program?

```
program Exercise2(input, output);
var A, B: integer;
procedure Arthur(var X, Y: integer);
begin{Arthur}
  X := 2; X := X + 1; Y := 2*X
end; {Arthur}
begin{Program}
  A := 4; B := 5;
  Arthur(A, B);
  writeln(A, B);
  A := 4; B := 5;
  Arthur(B, A);
  writeln(A, B)
end. {Program}
```

3. What is the output of the following program?

```
program Exercise3(input, output);
procedure Proced1;
begin {Proced1}
  write('One ')
end; {Proced1}
procedure Proced2;
begin {Proced2}
  Proced1;
  write('Two ')
end; {Proced2}
procedure Proced3;
begin {Proced3}
  Proced2;
  writeln('Three ')
end; {Proced3}
begin {Program}
  Proced1; writeln;
  Proced2; writeln;
  Proced3; writeln
end. {Program}
```

4. What is the output of the following program?

```
program Exercise4(input, output);
var X, Y: integer;
procedure Tricky(var Y, X: integer);
begin{Tricky}
  writeln(X, Y)
end; {Tricky}
```

---



```

begin {Program}
  X := 1; Y := 2;
  Tricky(Y, X);
  Tricky(Y, Y);
  Tricky(X, Y)
end. {Program}

```

5. Write a procedure with one variable parameter of type integer that leaves the parameter unchanged if it is positive or zero and changes it to zero if its value is negative.
6. Write a procedure with one formal variable parameter, Ans, of type char. If the value of Ans is 'Y', the procedure changes it to 'y'; if the value is anything else, it is not changed.

---

## Value Parameters

An actual variable parameter is a variable that the procedure can access or change. Variable parameters can be used to bring information into the procedure and/or pass information out of the procedure to the rest of the program. There is another class of parameter called *value parameters*. Value parameters are “one-way” parameters; they can be used to supply information to a procedure, but they cannot be used to get information out of a procedure. On the positive side, value parameters allow us to use more complicated expressions as the actual parameters in a procedure call. The notion is best introduced by means of an example.

*one-way  
data*

Suppose we want to write a procedure that will output the area of a rectangle to the screen. In this case, the procedure will have two parameters, one for the length of the rectangle and one for the width. Our first attempt at writing the procedure might be as follows:

```

procedure OutputArea1(var Length, Width: integer);
begin{OutputArea1}
  writeln('A rectangle of dimensions:');
  writeln(Length, ' by ', Width, ' inches');
  writeln('Has area ', Length * Width, ' square inches.')
end; {OutputArea1}

```

This will work, but in some situations it is inconvenient. If the length and width are stored in variables, say, X and Y, then to output the area of the rectangle, the following procedure call will do nicely:

```
OutputArea1(X, Y)
```

Suppose, however, that we wish to output the area of a rectangle that is 4 inches long by 3 inches wide. Since an actual variable parameter must be a variable, we will first have to set two variables equal to 4 and 3 and then use the variables as the actual parameters. That is unfortunate. It would be easier and cleaner if we could write the following

---

expression and then have the computer substitute 4 for the formal parameter `Length` and 3 for the formal parameter `Width`.

```
OutputArea1 (4, 3)
{Will not work with a variable parameter.}
```

As the comment indicates, this simply will not work with variable parameters. To make the above procedure call work, we must change `Length` and `Width` to value parameters.

*syntax for  
formal  
value  
parameters*

*Formal value parameters* are listed in the procedure heading just like formal variable parameters are, except that a formal value parameter is not preceded by the word `var`. To make `Length` and `Width` value parameters in our sample procedure, all we need to do is to omit the `var`, resulting in the following procedure heading:

```
procedure OutputArea2 (Length, Width: integer);
```

The rest of the procedure is unchanged. The complete procedure using formal value parameters is shown in in Figure 4.7.

*actual  
value  
parameters*

When a procedure with formal value parameters is called, it must be supplied with one *actual value parameter* to correspond to each formal value parameter. The relationship between formal and actual *value* parameters is similar to that between formal and actual *variable* parameters. The formal value parameter serves as a blank to be filled in, and the actual value parameter provides the item that fills in the blank. However, unlike an actual *variable* parameter, an actual *value* parameter is not simply “plugged in” as is. It is first evaluated and its *value* is then plugged in for the formal parameter. For example, if the actual value parameter is a variable `X` whose value is 5, then it is the 5 that is used rather than the `X`. After the values of the formal value parameters have been set, the statements in the procedure declaration are executed. In the next chapter, we will give more details about how this substitution is actually carried out, but this simple description is adequate to explain most normal uses of value parameters: The computer evaluates the actual value parameter to obtain a value and then substitutes this value into the formal value parameter.

This substitution mechanism explains why value parameters are one-way parameters. Since an actual value parameter is just a value, it cannot be changed. A procedure can change the value of a variable `X`, but it cannot change the value of the number 5. To emphasize this point with an example, suppose that `X` is used as an actual *value* parameter and that `X` has the value 5. In this case the procedure gets only the 5. It has no access to `X` and so it cannot possibly change `X`. Having discussed the largest restriction on value parameters, let us now discuss their biggest advantage.

*expressions  
as  
parameters*

Since only the value of an actual value parameter is used, the actual value parameter can be any expression that evaluates to the specified type. This means that the actual value parameter can be a variable, but it might instead be a constant or an arithmetic expression or anything that evaluates to a value of the correct type. Hence, with `OutputArea2`, all of the following sample procedure calls are allowed (`X` and `Y` are variables of type `integer`):

```
OutputArea2 (X, Y);
OutputArea2 (7, 8);
OutputArea2 (X + 7, Y mod X)
```

**Program**

```

program Value(input, output);
{Sample use of value parameters.}
var X: integer;

procedure OutputArea2(Length, Width: integer);
begin{OutputArea}
  writeln('A rectangle of dimensions:');
  writeln(Length, ' by ', Width, ' inches');
  writeln('has area ', Length * Width, ' square inches.')
end; {OutputArea}

begin{Program}
  OutputArea2(4, 3);
  X := 5;
  OutputArea2(X, X + 1)
end. {Program}

```

**Output**

```

A rectangle of dimensions:
  4 by      3 inches
has area   12 square inches.
A rectangle of dimensions:
  5 by      6 inches
has area   30 square inches.

```

**Figure 4.7**  
**Procedure with**  
**value parameters.**

---

## What Kind of Parameter to Use

Deciding whether to use variable or value parameters is fairly easy. If the procedure is supposed to provide information to some other part of the program, then use a variable parameter (or parameters) and have the procedure set the value(s) of the variable parameter(s). If the parameter is being used only to give information to the procedure rather than to get information out of the procedure, then a value parameter can and probably should be used. Some of the differences between the two kinds of parameters are listed in Figure 4.8.

Choosing what parameters as well as what kinds of parameters a procedure will use should be a trivial matter provided you have designed your algorithm with attention to the flow of information between tasks. When you divide the task of your program into subtasks, make two lists for each subtask. One list is the list of all the information that will be needed to accomplish the task; these parameters are sometimes called *in* parameters. The other list should show all the information that must be provided by this subtask to other subtasks or to the calling program; these parameters are sometimes called *out* parameters. The value parameters are easy to derive from the list of *in* parameters. The variable parameters are easy to derive from the list of *out* parameters. If

*data flow*



	Variable Parameter	Value Parameter
<i>var</i> in formal parameter list	YES	NO
Can be used to pass information to the procedure	YES	YES
Can be used to pass information from the procedure back to the calling program or procedure	YES	NO
Expressions allowed in the actual parameter list	NO (just variables)	YES

**Figure 4.8**  
Differences  
between variable  
and value  
parameters.

*summary  
of the  
kinds of  
parameters*

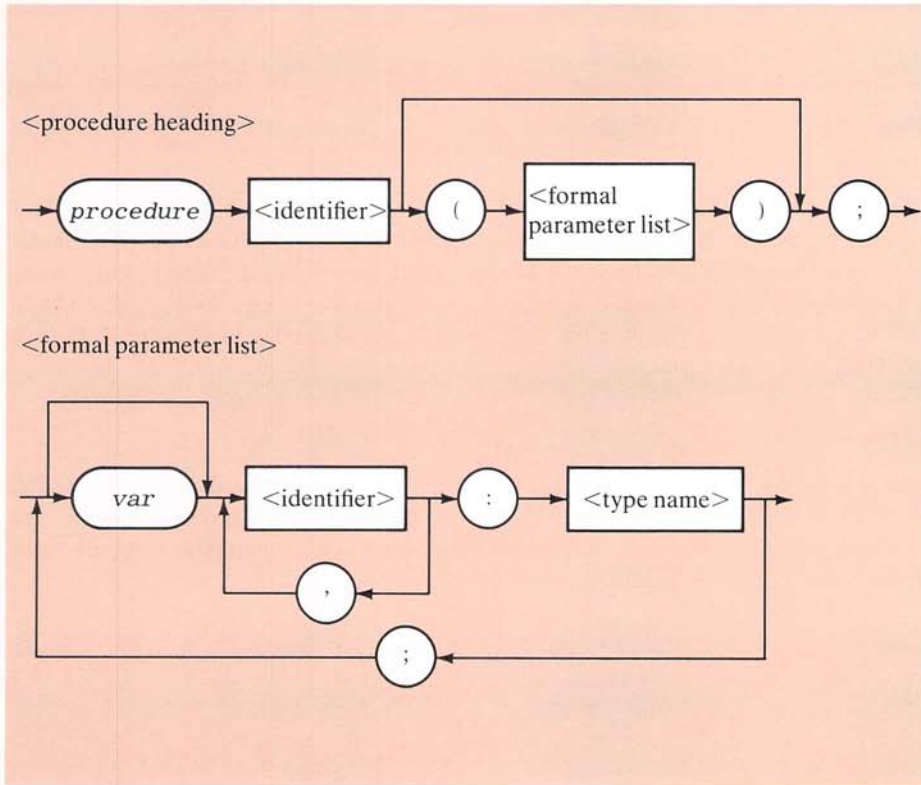
a quantity appears on both lists, then it must be a variable that will be changed by the procedure, and so it must be a variable parameter.

There are four kinds of parameters: formal variable parameters, actual variable parameters, formal value parameters, and actual value parameters. The long names convey meaning about how the parameters are used. The way to think of parameters is in a two-step process: First, the parameter is either *formal* or *actual*; second, it is either *variable* or *value*. These two two-way distinctions yield the four possible kinds of parameters. A *formal parameter* appears in a procedure declaration and is changed when the procedure is called. The *actual parameter* governs the change, and that change is always some sort of substitution of the actual parameter for the formal parameter. A formal parameter and its corresponding actual parameter are either both called variable parameters or both called value parameters. The distinction between value and variable parameters refers to the manner in which the substitution is performed. In the case of *variable parameters*, the actual parameter is a variable and is literally substituted for the formal parameter. In the case of *value parameters*, the actual parameter may be a more complicated expression, and it is the value of the expression that is substituted for the formal parameter.

Most programming languages have a distinction similar to the one in Pascal between value and variable parameters. All programming languages with facilities for parameters employ the distinction between formal and actual parameters.

## Mixed Parameter Lists

You may use any number of parameters in a procedure, and they may be any combination of variable and value parameters. You simply list them all in the procedure heading. For example, one procedure heading might be



**Figure 4.9**  
Syntax for a  
procedure heading.

*procedure* Sample(W: real; var X: real; Y, Z: char);

Each formal parameter has a type associated with it. In the parameter list, formal variable parameters are preceded by the word *var*. Formal value parameters are indicated by the absence of the *var*. In the sample heading, X is a variable parameter; W, Y, and Z are value parameters. The various formal parameters are separated by semicolons. When two or more consecutive parameters are of the same type and are either all value or all variable parameters, then you can combine their type specifications in the way illustrated by Y and Z in the sample heading. The details are summarized by the syntax diagram in Figure 4.9.

In the procedure call, the actual parameters are given in parentheses after the procedure name; they must correspond in type to the formal parameters given in the procedure heading. For example, in the procedure call

Sample(2.5, A, 'B', C)

The variable A must have been declared to be of type *real*, and the variable C must have been declared to be of type *char*. All actual variable parameters must be variables, and so in the above example A must be a variable. Since the other parameters in this example are value parameters, they may be variables, constants, or more complicated expressions.

## Case Study

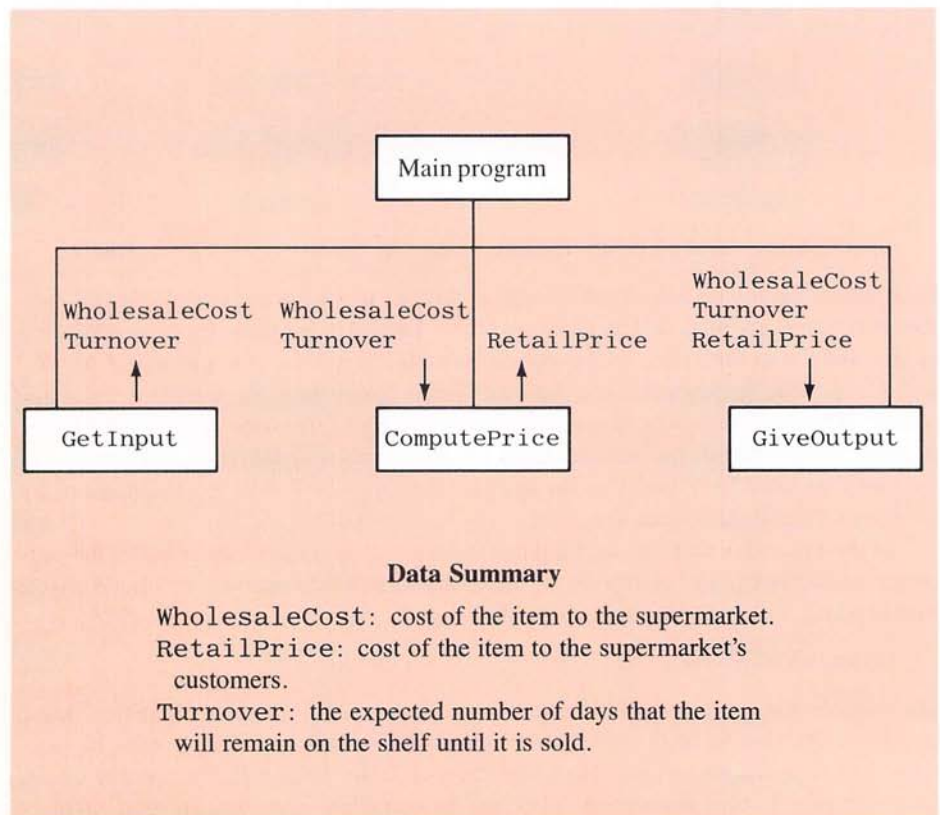
### Supermarket Pricing

#### Problem Definition

We have been commissioned by the Super-Duper supermarket chain to write a program that will determine the retail price of an item, given suitable input. Their pricing policy is that any item that is expected to sell in less than one week is marked up 5% and any item that is expected to stay on the shelf for at least one week is marked up 10% over the wholesale price.

#### Discussion

Like many other programming tasks, this one breaks down into three main subtasks: input the data, perform a computation, and output the results. The decomposition into subtasks, as well as the flow of data and the meaning of the variables to be used are shown in the data flow diagram in Figure 4.10.



**Figure 4.10**  
Data flow diagram  
for supermarket  
pricing.



The arrows in the data flow diagram indicate the direction of data flow. If an arrow is pointing up out of a box, then the procedure represented by that box will change the quantities associated with that arrow, and so the corresponding procedure parameter will have to be a variable parameter. If a quantity is associated with an arrow pointing down into the box and is not associated with any arrow pointing up out of the box, then we will use a value parameter that takes the value of the quantity. So, for example, the procedure call to the second procedure will read

```
ComputePrice(WholesaleCost, Turnover, RetailPrice)
```

The first two parameters will be value parameters and the last one must be a variable parameter, and so the procedure heading will read as follows:

```
procedure ComputePrice(Cost: real;
                       Time: integer; var Price: real);
```

The algorithm for this procedure is straightforward:

**ALGORITHM**

```
if Time < 7 then
    Price := Cost + 5% of Cost
else
    Price := Cost + 10% of Cost
```

The complete program is displayed in Figure 4.11.

### Program

```
program Pricing(input, output);
{Determines the retail price of an item according to the
pricing policies of the Super-Duper supermarket chain.}
const LowMarkup = 0.05; {5%}
      HighMarkup = 0.10; {10%}
      Threshold = 7; {Use HighMarkup if do not expect to sell in under 7 days.}
      Width = 5; {Field width for prices.}
var WholesaleCost, RetailPrice: real;
    Turnover: integer; {Expected time on shelf, expressed in days.}

procedure GetInput(var Cost: real; var Time: integer);
begin{GetInput}
    writeln('Enter the wholesale cost of the item: ');
    readln(Cost);
    writeln('Enter expected number of days until sold: ');
    readln(Time)
end; {GetInput}

procedure ComputePrice(Cost: real;
                       Time: integer; var Price: real);
{Determines the retail Price of an item, given the
wholesale Cost and the expected time until it is sold.}
```

**Figure 4.11**  
**Supermarket**  
**pricing program.**

```

begin{Compute}
  if Time < Threshold then
    Price := (1 + LowMarkup) * Cost
  else
    Price := (1 + HighMarkup) * Cost
end; {Compute}

procedure GiveOutput(Cost: real; Time: integer; Price:real);
begin{GiveOutput}
  writeln('Wholesale cost = $', Cost:Width:2);
  writeln('Expected time until sold = ', Time, ' days');
  writeln('Retail price = $', Price:Width:2)
end; {GiveOutput}

begin{Program}
  writeln('This program determines');
  writeln('retail prices for stock items. ');
  GetInput(WholesaleCost, Turnover);
  ComputePrice(WholesaleCost, Turnover, RetailPrice);
  GiveOutput(WholesaleCost, Turnover, RetailPrice)
end. {Program}

```

#### Sample Dialogue

```

This program determines
retail prices for stock items.
Enter the wholesale cost of the item:
1.21
Enter expected number of days until sold:
5
Wholesale cost = $ 1.21
Expected time until sold =    5 days
Retail price = $ 1.27

```

**Figure 4.11**  
(continued)

---

## Case Study

---

### Change Program with Procedures

#### Problem Definition

We will redesign the change program from Chapter 2 so that it uses procedures for subtasks. The program will accept an amount between 1 and 99 and will output a combination of quarters, dimes, and pennies that total to that amount.

---

## Discussion

For reference, let us summarize our previous analysis of subtasks and data flow for this problem. We need a variable `Amount` to hold the total amount of money. We also need variables to hold the number of each coin, as well as a variable `AmountLeft` to hold the amount left to be given out as we proceed down the list of coins from quarters to dimes to pennies computing the number of each coin to be given out. The breakdown into subtasks is summarized in the following pseudocode:

```
begin{Program}
  1. InputAmount: Input the amount and store it in the variable Amount;
  2. AmountLeft := Amount;
  3. ComputeChange: Compute a combination of quarters, dimes, and pennies
     whose value equals AmountLeft;
  4. OutputCoins: Output Amount and the number of each coin
end. {Program}
```

All subtasks except subtask 2 will be implemented as procedures. The data flow diagram in Figure 4.12 shows the flow of data and can be used to determine which parameters need to be variable parameters and which may be value parameters. All of the parameters for `InputAmount` and `ComputeChange` must be variable parameters. We know this because the variables listed above the `InputAmount` and `ComputeChange` boxes all have arrows that point up out of the box, meaning that the parameters may be changed in the procedures. The variables above the procedure `OutputCoins` have no arrows pointing up out of the box, and so the parameters for this procedure may be value parameters.

Subtask 3 (`ComputeChange`) is further subdivided as follows; we will implement each of these smaller subtasks as a procedure that uses all variable parameters:

```
begin{ComputeChange}
  2a. ComputeQuarters: Compute the maximum number of quarters in
     AmountLeft and decrease AmountLeft by the total value of the quarters;
  2b. ComputeDimes: Compute the maximum number of dimes in
     AmountLeft and decrease AmountLeft by the total value of the dimes;
  2c. ComputePennies: Compute the number of pennies in AmountLeft
end {ComputeChange}
```

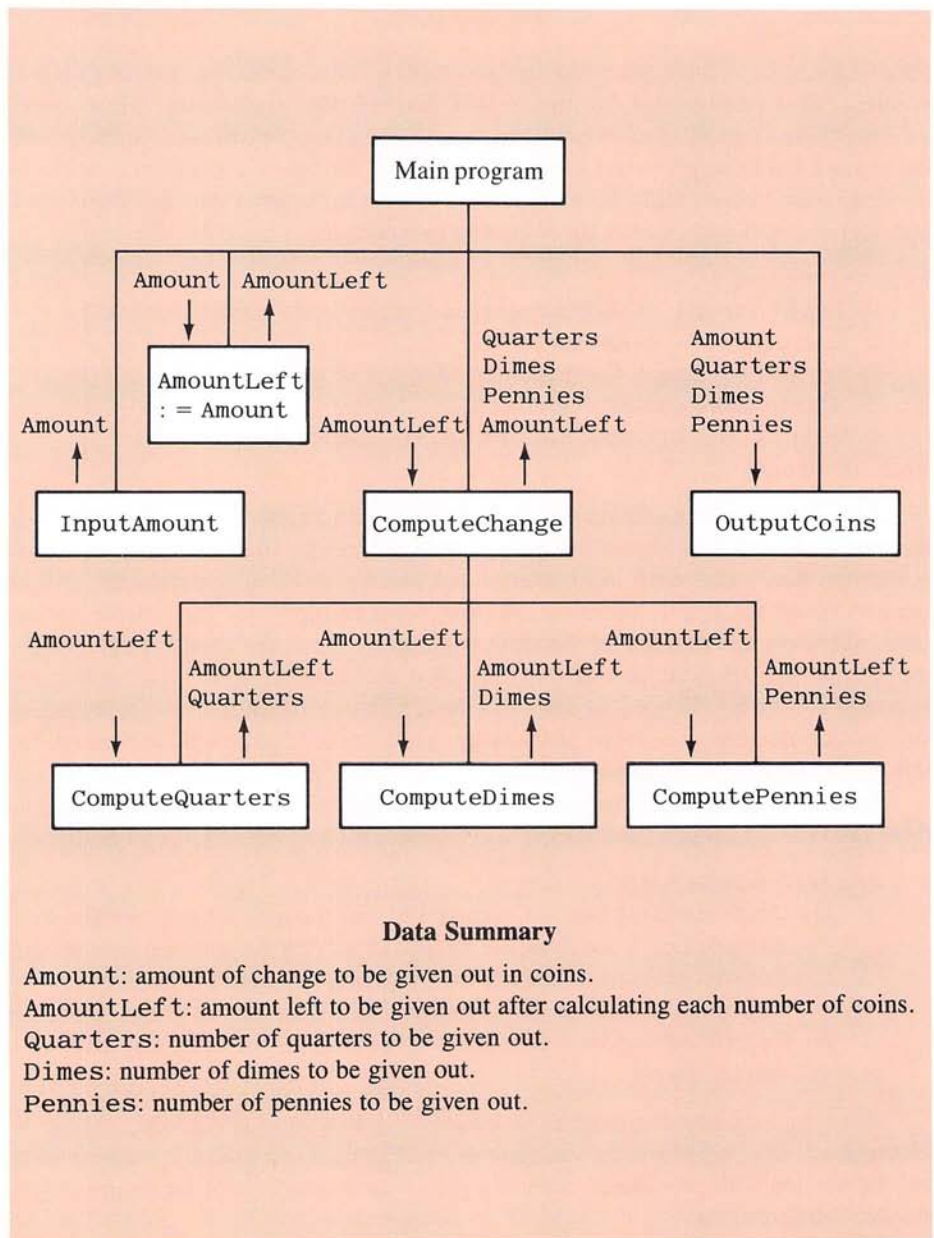
The completed program is given in Figure 4.13. It is equivalent to the program in Figure 2.13. The two programs will carry on exactly the same dialogue with the user, but the version with procedures explicitly shows the breakdown of the program into smaller subalgorithms.

We first wrote this change-making program without procedures and then rewrote it using procedures. We did this because we did not know about procedures when we first wrote the program. Normally, each subtask is translated into a procedure before it is used in a Pascal program. The stepwise refinement of tasks into subtasks and the writing of procedures go hand in hand.

## ALGORITHM

*procedures  
for subtasks*





**Figure 4.12**  
**Data flow diagram**  
**for change**  
**program.**

### Program

```
program Change2(input, output);  
{Outputs the coins used to give an amount between 1 and 99 cents.}  
var Amount, AmountLeft,  
    Quarters, Dimes, Pennies: integer;  
  
procedure InputAmount(var A: integer);  
{Fills the variable A with a value between 1 and 99 (inclusive).}  
begin{InputAmount}  
    writeln('Enter an amount of change');  
    writeln('from 1 to 99 cents:');  
    readln(A)  
end; {InputAmount}  
  
procedure ComputeQuarters(var Q, ALeft: integer);  
{Sets Q equal to the maximum number of quarters in ALeft cents.  
ALeft is decreased by the value of the quarters.}  
begin{ComputeQuarters}  
    Q := ALeft div 25;  
    ALeft := ALeft mod 25  
end; {ComputeQuarters}  
  
procedure ComputeDimes(var D, ALeft: integer);  
{Sets D equal to the maximum number of dimes in ALeft cents.  
ALeft is decreased by the value of the dimes.}  
begin{ComputeDimes}  
    D := ALeft div 10;  
    ALeft := ALeft mod 10  
end; {ComputeDimes}  
  
procedure ComputePennies(var P, ALeft: integer);  
{Sets P equal to (the number of pennies in) ALeft (cents) and sets ALeft equal to zero.}  
begin{ComputePennies}  
    P := ALeft;  
    ALeft := 0  
end; {ComputePennies}  
  
procedure ComputeChange(var ALeft, Q, D, P: integer);  
{Sets the value of Q, D, and P to a number of quarters, dimes, and pennies  
that total to ALeft cents. (The value of ALeft is changed to zero.)}  
begin{ComputeChange}  
    ComputeQuarters(Q, ALeft);  
    ComputeDimes(D, ALeft);  
    ComputePennies(P, ALeft)  
end; {ComputeChange}
```

**Figure 4.13**  
**Change-making**  
**program using**  
**procedures.**

```
procedure OutputCoins(A, Q, D, P: integer);  
{Outputs the values of all parameters. Includes a heading stating that A cents is  
equal to the total of Q quarters, plus D dimes, plus P pennies.}  
begin{OutputCoins}  
  writeln(A, ' cents can be given as:');  
  writeln(Q, ' quarters');  
  writeln(D, ' dimes and');  
  writeln(P, ' pennies')  
end; {OutputCoins}  
  
begin{Program}  
  InputAmount(Amount);  
  AmountLeft := Amount;  
  ComputeChange(AmountLeft, Quarters, Dimes, Pennies);  
  OutputCoins(Amount, Quarters, Dimes, Pennies)  
end. {Program}
```

#### Sample Dialogue

```
Enter an amount of change  
from 1 to 99 cents:  
96  
96 cents can be given as  
3 quarters  
2 dimes and  
1 pennies
```

**Figure 4.13**  
(continued)

## Pitfall

### Incorrectly Ordered Parameter Lists

If the order of an actual parameter list does not correspond to the desired substitution pattern, the program will not work correctly. The computer substitutes the first actual parameter for the first formal parameter, the second actual parameter for the second formal parameter, and so forth. It does not care about any mnemonic matches such as Quarters for Q or Pennies for P. To see what can happen when parameter lists are ordered incorrectly, let us once again consider the change-making program in Figure 4.13. Suppose we had mistakenly ordered the actual parameters in the procedure call to OutputCoins as follows:

```
OutputCoins(Amount, Pennies, Dimes, Quarters)
```



If this were the case, the last portion of the output would change to

```
96 cents can be given as
1 quarters
2 dimes and
3 pennies
```

If the formal and actual parameters do not match in type, the computer will give an error message. However, in cases such as this one, in which all the parameters are of the same type, a misordering of the parameter list will simply produce an incorrect result. Parameters “differently arranged have a different meaning, and meanings differently arranged have different effects.”

---

## Generalizing Procedures

Once you have broken a problem down into subtasks, it is a good idea to look for general procedures that can solve more than one subtask merely by varying some of their parameters. For example, notice in the change-making program in Figure 4.13 that the procedure for calculating the numbers of quarters and the one for computing the number of dimes are very similar. They each contain one application of *div* to find the number of coins and one application of *mod* to find the amount left after giving out that many coins. For quarters, the calculation is

```
Q := ALeft div 25;
ALeft := ALeft mod 25
```

For computing dimes, it is

```
D := ALeft div 10;
ALeft := ALeft mod 10
```

These two calculations differ in just two places: in one case a *Q* is used on the left-hand side of the assignment operator, and in the other case a *D* is used; in one case a 25 is used, and in the other case a 10 is used. By using two formal parameters, we can obtain a single procedure that can perform either of these computations. If we call these parameters *Number* and *CoinValue*, the more general calculation reads as follows:

```
Number := ALeft div CoinValue;
ALeft := ALeft mod CoinValue
```

The complete procedure, called *ComputeCoins*, as well as the procedure *ComputeChange* that calls it, are shown in Figure 4.14. We have extended the procedure *ComputeChange* so that it also uses nickels. With the general procedure *ComputeCoins* at our disposal, this is easy to do. We could just as easily add half-dollar coins; such additions do not significantly complicate the program. If efficiency is

---

```

procedure ComputeCoins(CoinValue: integer;
                       var Number, ALeft: integer);
{Sets Number equal to the maximum number of coins in ALeft cents, where each coin
has value CoinValue cents. ALeft is decreased by the value of the coins.}
begin{ComputeCoins}
    Number := ALeft div CoinValue;
    ALeft := ALeft mod CoinValue
end; {ComputeCoins}

procedure ComputeChange(var ALeft, Q, D, N, P: integer);
{Sets the value of Q, D, N, and P to a number of quarters, dimes, nickels, and pennies
that total to ALeft cents. (The value of ALeft is changed to zero.)}
begin{ComputeChange}
    ComputeCoins(25, Q, ALeft);
    ComputeCoins(10, D, ALeft);
    ComputeCoins(5, N, ALeft);
    ComputeCoins(1, P, ALeft)
end; {ComputeChange}

```

**Figure 4.14**  
General procedure  
for number of  
coins.

a major issue, the last call to `ComputeCoins` can be replaced by the following simpler way of setting the value of `P`.

```
P := AmountLeft
```

Notice that in order to make the general procedure `ComputeCoins` work in a variety of cases, we needed to add one additional parameter beyond those needed by the specialized procedures, such as `ComputeQuarters`. One often needs to add one or more parameters to a procedure design in order to fit it to a particular use.

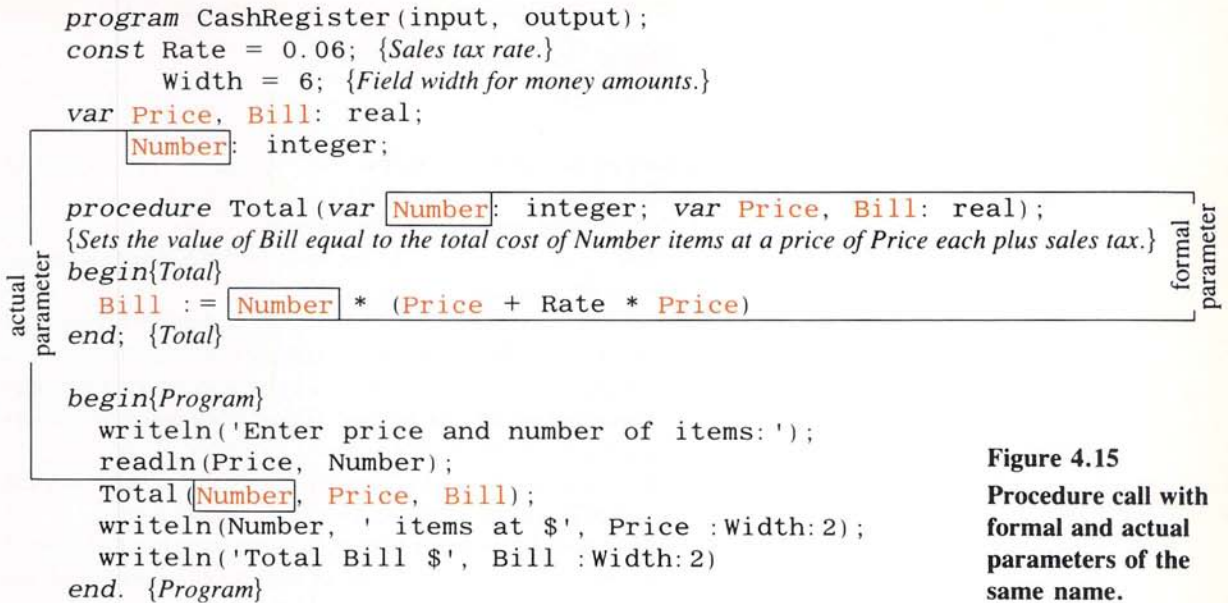
As was true in this example, it is often the case that the generalization is not apparent until after you have written a few of the specialized procedures that it will replace. If a generalized procedure turns out to be sensible, then do not hesitate to discard the old procedures and replace them with the new one. The work done on the old procedures is not wasted; it helped you to find the better solution.

---

## Choosing Parameter Names

Procedures are self-contained units that are best designed separately from the rest of the program. On large programming projects, a different programmer may be assigned to write each procedure. The programmer should choose the most meaningful names for formal parameters that he or she can find. The actual parameters that will be substituted for the formal parameters should also be given meaningful names, often chosen by someone else. When you are proceeding in this way, it is likely that some or

---



```

program CashRegister(input, output);
const Rate = 0.06; {Sales tax rate.}
      Width = 6; {Field width for money amounts.}
var Price, Bill: real;
    Number: integer;

procedure Total(var Number: integer; var Price, Bill: real);
{Sets the value of Bill equal to the total cost of Number items at a price of Price each plus sales tax.}
begin{Total}
    Bill := Number * (Price + Rate * Price)
end; {Total}

begin{Program}
    writeln('Enter price and number of items:');
    readln(Price, Number);
    Total(Number, Price, Bill);
    writeln(Number, ' items at $', Price :Width:2);
    writeln('Total Bill $', Bill :Width:2)
end. {Program}

```

**Figure 4.15**  
**Procedure call with**  
**formal and actual**  
**parameters of the**  
**same name.**

all pairs of formal and actual parameters will have the same name. This is perfectly acceptable. Figure 4.15 shows a rewritten version of the program in Figure 4.5. In the rewritten version, the formal and actual parameters are given identical names. Technically speaking, the formal parameter *Bill* and the actual parameter *Bill* are two different objects that just happen to have the same name. The computer will even substitute the actual parameter *Bill* for the formal parameter *Bill*. The programs in Figures 4.5 and 4.15 are handled in the same way by the compiler and act exactly alike when run.

This unrestricted choice of parameter names is an important part of the top-down, divide-and-conquer method of writing programs. When you are writing a procedure, you should be concentrating on the procedure and what it does. You should not have to think about what names are used for variables in the main program.

After you become comfortable with parameters, you will naturally tend to give formal and actual parameters the same names. For your first few programs, however, parameters may be easier to understand if you use different names for formal and actual parameters.

---

Words differently arranged have a different meaning, and  
meanings differently arranged have different effects.

---

*Blaise Pascal*



## Summary of Problem Solving and Programming Techniques

- When designing programs by the top-down method, you should normally implement subtasks as procedures.
- Parameters are used to pass information between procedures.
- The first steps in designing a procedure are to decide what task it is supposed to accomplish and what information it needs.
- Analyzing the flow of information into and out of a procedure will make it clear what parameters are needed and whether they should be variable or value parameters.
- Variable parameters can be used to give information to a procedure or to get information out of a procedure. Value parameters can be used to give information to a procedure, but they cannot be used to get information out of a procedure.
- Constants and complicated expressions, as well as variables, can be used as actual value parameters. Only variables can be used as actual variable parameters.
- Once a procedure is written and debugged, a user should not have to know how it works. The procedure heading and accompanying comment should explain what the procedure does, and that should suffice to use the procedure.
- Look for opportunities to combine two or more procedures into one more general procedure.

---

## Summary of Pascal Constructs

### procedure declaration

Syntax:

```
procedure <procedure name> (<formal parameter list>);  
begin  
    <statement>;  
    <statement>;  
    .  
    .  
    .  
    <statement>  
end;
```

Example:

```
procedure Check(Hours: integer; var Salary, Tax: real);  
begin  
    Salary := 12.50 * Hours;  
    Tax := 0.25 * Salary  
end;
```

---

The statements may be any Pascal statements and may contain the formal parameters. (See the next two entries in this summary.)

### procedure heading

Syntax:

```
procedure <procedure name> (<formal parameter list>);
```

Examples:

```
procedure Check (Hours: integer; var Salary: real;  
                var Tax: real);  
procedure Check (Hours: integer; var Salary, Tax: real);
```

The procedure heading is the first thing in a procedure declaration. The <procedure name> can be any identifier other than a reserved word. The <formal parameter list> is a list of identifiers that will serve as formal parameters. Each formal parameter is followed by a colon and its type. The formal variable parameters are prefaced by *var*; formal value parameters are not. The parameters are separated by semicolons. Consecutive formal parameters of the same kind (that is, ones that are the same type and that are either all value or all variable parameters) may (optionally) be grouped together and separated by commas. The above two examples are equivalent.

### formal parameter

A formal parameter appears in a procedure declaration and is changed when the procedure is called. The change is a substitution of either a variable or a value for the formal parameter. The formal parameters for a procedure are listed in the procedure heading.

### procedure call

Syntax:

```
<procedure name> (<actual parameter list>)
```

Example:

```
Check (40, PayForJoe, TaxFromJoe)
```

The <actual parameter list> contains the actual parameters separated by commas. There must be exactly as many actual parameters as there are formal parameters. The first actual parameter corresponds to the first formal parameter, the second actual parameter corresponds to the second formal parameter, and so forth. Corresponding formal and actual parameters must agree in type. If a formal parameter is a variable parameter (that is, one that is prefaced by *var*), then the corresponding actual parameter must be a variable. If a formal parameter is a value parameter (that is, one that is not prefaced by *var*), then the corresponding actual parameter may be anything that evaluates to the type of the formal parameter. When a procedure is called, some kind of substitution of actual parameters for the corresponding formal parameters is made, and then the procedure statements are executed. (See *variable parameter* and *value parameter*.)

**actual parameter**

The value or variable substituted for a formal parameter when a procedure is called. The actual parameters are listed in a procedure call as shown in the previous entry.

**variable parameter**

Formal and actual parameters come in pairs. The pair is either a pair of value parameters or a pair of variable parameters. If the pair is a pair of variable parameters, then a *var* is placed in front of the formal parameter in the formal parameter list of the procedure heading. If the pair is a pair of variable parameters, the actual parameter must be a variable; the formal parameter is a labeled blank, and when the procedure is called, the actual parameter is substituted for the corresponding formal parameter. Thus, the actual parameter may be changed.

**value parameters**

Formal and actual parameters come in pairs. The pair is either a pair of value parameters or a pair of variable parameters. If the formal parameter is not prefaced by *var* in the formal parameter list of the procedure heading, then the pair is a pair of value parameters. If the pair is a pair of value parameters, the actual parameter may be anything that evaluates to the type of the corresponding formal parameter. When the procedure is called, the value of the formal parameter is set equal to the value of the corresponding actual parameter. Thus, the actual parameter cannot be changed.

---

## Exercises

### Self-Test Exercises

7. Can a *variable* parameter be used to give information to a procedure? Can a *value* parameter be used to give information to a procedure? Can a *variable* parameter be used to get information out of a procedure? Can a *value* parameter be used to get information out of a procedure?

8. Which of the following are allowed as actual *value* parameters of type *integer*? (X is a variable of type *integer*.)

X      X + 1      abs (2\*X)      25

Which are allowed as actual *variable* parameters of type *integer*?

9. Can an actual *variable* parameter be a variable? Can an actual *value* parameter be a variable? Can an actual *variable* parameter be a constant? Can an actual *value* parameter be a constant?

10. What is the output of the following program?

```
program WatchIt(input, output);  
var X, Y: integer;
```

---



```
procedure Sam(X, Y: integer);  
begin{Sam}  
    writeln(X, Y)  
end; {Sam}  
begin{Program}  
    X := 1; Y := 2;  
    Sam(X, Y);  
    Sam(Y, X)  
end. {Program}
```

### Interactive Exercises

11. Type up and run the program in Figure 4.5. Then interchange the order of the formal parameters P2 and P3 in the procedure heading and run the program again.
12. Write a procedure to find the area of a rectangle and store the answer in a variable parameter called A. This procedure will have two value parameters, as in `OutputArea2` (Figure 4.7), plus the variable parameter A, for a total of three formal parameters. Embed this in a program and run the program.
13. Type up and run the program in Figure 4.15. Interchange the parameters `Bill` and `Price` in the procedure call (but not in the procedure heading) and run the program again.

### Programming Exercises

14. Write a procedure with three parameters, one value parameter of type `real` and two variable parameters of type `integer`. One variable parameter is set to the whole number part of the `real` value; the other is set to the value of the first digit after the decimal point.
15. Write a program that reads in a real number (representing that many inches) and then outputs the area of a square with sides of that length and the area of a circle with a diameter of that length. Use two procedures to compute the two areas.
16. Write a procedure that has two formal parameters, one for the radius of a circle and one for the circumference. Given a radius, the procedure computes the circumference of the circle and stores the answer in the parameter for the circumference. Embed this in a program to compute the circumference of a circle.
17. Write a program that converts dollars to Japanese yen or yen to dollars, depending on the user's desire. The user is asked which conversion he or she wants performed. If the desired conversion is yen to dollars, then the program reads in an amount in yen as well as the yen-to-dollar exchange rate. The program then outputs the equivalent amount in dollars and cents. If the user instead requests a conversion from dollars to yen, the roles of dollars and yen are interchanged. Use at least four procedures: one for input, one for output, one to convert from yen to dollars, and one to convert from dollars to yen. (If you wish, you can write one more general procedure in place of the last two mentioned. You may find it convenient to have two output procedures, one for each type of conversion.)

18. Write a program that writes “HELLO” to the screen, one letter at a time. Use four procedures for the four letters H, E, L, and O. Each letter should be a pattern of asterisks that is at least five times larger than the regular letters on your screen. Each letter of the word should be indented more than the previous letter so that the output looks something like the following, only larger:

```
  H
   E
    L
     L
      O
```

To display the next letter, the user presses the return key. (It may not all fit on the screen at once.)

19. Write a procedure that has one value parameter of type `char`. The procedure writes “YES” to the screen if the parameter value is ‘y’ and “NO” if it is anything else. Embed it in a test program.

20. Write two parameterless procedures, one to write the word YES and one to write the word NO to the screen in large letters made up of asterisks. The letters should be at least five times the normal size of letters on your screen. Use these procedures to redo the previous exercise to output larger words.

21. Write a procedure that computes the average and standard deviation of four scores. The standard deviation is defined to be the square root of the average of the four values  $(s_i - a)^2$ , where  $a$  is the average of the four scores  $s_1, s_2, s_3$ , and  $s_4$ . The procedure will have six parameters and will call two other procedures. Embed the procedure in a program to test it.

22. Write a program that asks the user to type in his or her height, weight, and age and then computes clothes sizes according to the following formulas: hat size = weight in pounds divided by height in inches, and all that multiplied by 2.9; sweater size (chest in inches) = height times weight, divided by 301 and then adjusted by adding  $\frac{1}{8}$  inch for every 10 years over age 30; waist in inches = weight divided by 5.7 and then adjusted by adding  $\frac{1}{10}$  inch for every 2 years over age 28. Use procedures for each calculation.

23. You have a choice of two different auto mechanics with different rate structures. One charges a flat fee of \$20 plus \$5.00 for each quarter-hour. The other charges \$18 for the first quarter-hour and \$5.75 for each quarter-hour after that. Any fraction of a quarter-hour counts as a full quarter-hour, and so 1.26 hours counts as six quarter-hours. Write a program that accepts the number of hours as input, displays the two charges, and announces which is cheaper. Use four or more procedures.

24. Redo (or do for the first time) Exercise 26 from Chapter 2. Use three procedures: one for input, one for output, and one to perform the calculation.

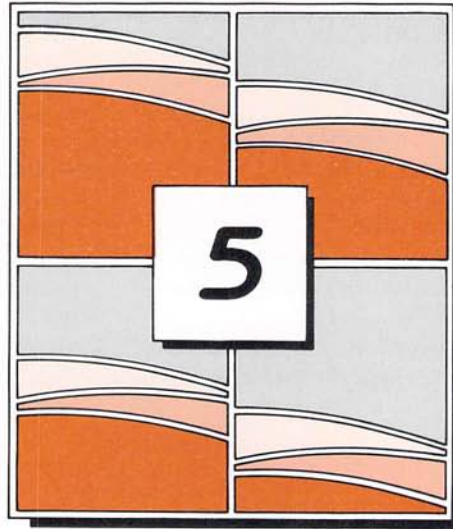
25. Redo (or do for the first time) Exercise 27 from Chapter 2. Use three procedures: one for input, one for output, and one to perform the calculation.

26. Redo (or do for the first time) Exercise 29 from Chapter 2. Use three procedures: one for input, one for output, and one to perform the calculation.

27. Redo (or do for the first time) Exercise 30 from Chapter 2. Use three procedures: one for input, one for output, and one to perform the calculation.
  28. Redo (or do for the first time) Exercise 31 from Chapter 2. Use three procedures: one for input, one for output, and one to perform the calculation.
  29. Redo (or do for the first time) Exercise 32 from Chapter 2. Use three procedures: one for input, one for output, and one to perform the calculation.
-







## ***Procedures for Modular Design***

“My memory is so bad,  
that many times I forget my own name!”  
*Miguel de Cervantes, Don Quixote*

## Chapter Contents

Local Variables  
Case Study—Grade Warnings  
Other Local Identifiers  
Pitfall—Use of Global Variables  
Self-Test Exercises  
Implementation of Value Parameters  
Pitfall—Inadvertent Local Variables  
Scope of an Identifier  
Case Study—Automobile Bargaining

Testing Procedures  
Top-Down and Bottom-Up Strategies  
Preconditions and Postconditions  
Case Study—Calculating Leap Years  
Summary of Problem Solving and Programming Techniques  
Summary of Pascal Constructs  
Exercises

**P**rocedures separate a program into smaller, and hence more manageable, pieces. In order to get the full benefit of this decomposition, the procedures must be self-contained units that are meaningful outside the context of any particular program. A program that is built out of such self-contained procedures is often said to have a *modular* design. In this chapter we show how a procedure can be made self-contained by providing it with its own independent set of variables. After introducing these variables, known as *local variables*, we then use the construct to give examples of modular design.



## Local Variables

In order to introduce the notion of a local variable, we will consider a simple but frequently occurring task and design a procedure to accomplish this task.

Suppose we wish to write a procedure to interchange the values of two variables of type `integer`. The procedure heading will be

```
procedure Exchange (var X, Y: integer);
```

The body of the procedure presents more of a problem. An obvious but incorrect solution to try is

```
X := Y; Y := X
```

This sets the new value of `X` to the old value of `Y`, as desired. But then it sets the new value of `Y` to the *new* value of `X`, and so leaves `Y` unchanged. (If this seems unclear, plug in some values and see what happens.) What we need to do is to save the original value of `X` before we change the value of `X`. We can then use that saved value to set the value of `Y`. What we need is another variable to hold this saved value temporarily. Let us call this extra variable `Temp`. The correct procedure now reads:

```
procedure Exchange (var X, Y: integer);
{Interchanges the values of X and Y.}
begin{Exchange}
  Temp := X;
  X := Y;
  Y := Temp
end; {Exchange}
```

This procedure will work nicely except for one annoying detail. Because the procedure `Exchange` changes the value of `Temp`, we must remember not to use the variable `Temp` for anything else. This is most unfortunate. The whole idea of top-down design and procedures is to break big tasks into smaller tasks. We make no headway in this direction if, when designing a procedure, we need to remember the details of how all the other procedures work. Once we get a procedure to work, we should only need to remember what it does, not how it does it. The principle of procedural abstraction that we advocated in the last chapter states that the procedure heading and one explanatory comment should suffice to use the procedure. Therefore, the following two lines should be all we need to know in order to use the procedure safely and effectively:

```
procedure Exchange (var X, Y: integer);
{Interchanges the values of X and Y.}
```

Clearly, these two lines are not all we need to remember. We must also remember that the procedure changes the value of the variable `Temp`.

Ideally, we should not have to remember anything about `Temp`, not even that it was used. What we need is a special version of the variable `Temp` that exists only for the duration of the procedure call. Fortunately, Pascal and many other programming languages allow such variables. They are called “local variables.”

A *local variable* is a variable that is declared within a procedure. Variables that are

*Exchange  
example*

*procedural  
abstraction*

*local  
variables*

*global  
variables*

declared for the entire program are called *global variables*. All the variables we have used until now are global variables. Local variables are declared just like global variables except that the declaration is placed inside the procedure declaration, between the procedure heading and the *begin* that marks the start of the procedure body. A local variable is meaningful only inside the procedure. No statement outside of the procedure body may reference a local variable.

The program in Figure 5.1 includes the procedure *Exchange* with *Temp* declared as a local variable. A *writeln* has been added to the procedure in order to show the value of the local variable. Although the procedure has numerous useful applications, this particular program does not have any applications. It is just an example to illustrate how global and local variables work.

*interpreting  
local  
variables*

The program in Figure 5.1 has two variables called *Temp*: one global variable declared for the entire program and one local variable declared for the procedure *Exchange*. What is the relationship between these two different variables? They share the same name, but they are two totally different variables. The computer does manage to keep track of these two different variables even though they have but one name be-

### Program

```

program Sample(input, output);
var A, B, Temp: integer;

procedure Exchange(var X, Y: integer);
{Interchanges the values of X and Y.}
var Temp: integer;
begin{Exchange}
    Temp := X;
    X := Y;
    Y := Temp;
    writeln(A, B, Temp)
    {This writeln is here to illustrate the concept of a
     local variable. It would not normally be included.}
end; {Exchange}

begin{Program}
    A := 1;
    B := 2;
    Temp := 3;
    writeln(A, B, Temp);
    Exchange(A, B);
    writeln(A, B, Temp)
end. {Program}

```

### Output

1	2	3
2	1	1
2	1	3

**Figure 5.1**  
Procedure with a  
local variable.

tween them. It is as if the computer acted in the following way: When a procedure is called, the computer checks to see whether there are any local variables, such as Temp in the procedure Exchange. If there are, it looks to see whether there are also any global program variables of the same name as one of these local variables; in this case, there is one such variable. The computer then saves the value of the global variable. The global variable becomes inactive, and the name is given to the local variable. The computer then executes the procedure call using the Pascal identifier as the name of the local variable. During the execution of the procedure, the shared name always refers to the local variable. When the procedure call is completed, the Pascal name is given back to the global variable, which still has its saved value, and from that point on there is no way to refer to the local variable. All the changes made to the local variable have no effect on the global variable of the same name.

In the procedure declaration in Figure 5.1, the identifier Temp on the left-hand side of the assignment operator names the local variable, not the global variable. Hence, the global variable is not changed. This is why the third output line is 2 1 3. If Temp had not been declared as a local variable, that is, if the line

```
var Temp: integer;
```

had been omitted from the procedure declaration, then the output instead would be

```
1 2 3
2 1 1
2 1 1
```

Because the computer handles local variables in this way, we can design procedures using local variables and not even bother to remember which identifiers we used for the local variables. If, outside the procedure, we reuse those identifiers to mean something else, the computer will know we mean something else.

*example*

---

## Case Study

---

### Grade Warnings

#### Problem Definition

We want to design a program that can be used early in the school term to let students know whether they are making satisfactory progress in a class. It is presumed that two tests are sufficient to obtain a meaningful early indication of progress, and so the program will read two test scores and output an answer telling whether or not the average of the two scores is passing. Since there is some danger that a student might use the program at the wrong time, the program will tell the student to see the instructor if the student has not taken exactly two tests.

#### Discussion

The program will simply ask the student how many tests he or she has taken. If the student has taken any number other than two tests, the program tells the student to see the instructor. The main outline of the program is

*ALGORITHM*

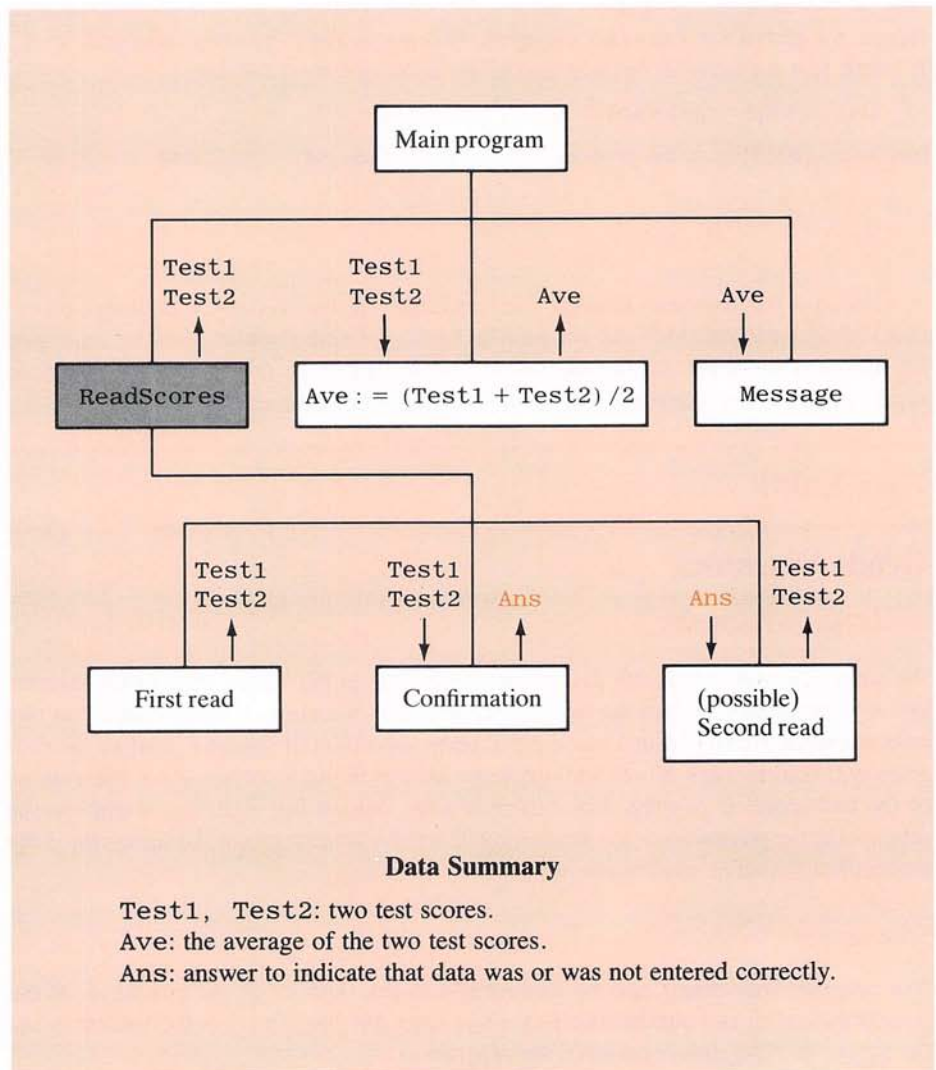


```

if the student has not taken exactly two tests then
  writeln('See the instructor.')
else
  begin
    Read in the two test scores.
    Compute the average.
    Output a suitable message.
  end

```

The two test scores will be read in by a procedure called `ReadScores`. Because the student might make a mistake in entering the test scores, the procedure will give the student two chances to enter them. It will read the two scores, display them, and then



**Figure 5.2**  
 Data flow diagram  
 for the case of  
 exactly two tests.

**Program**

```
program Warning(input, output);  
{Tells students if their first two tests represent passing work.}  
const Passing = 70;  
      Width = 7; {Field width for percentages = 5 + Frac.}  
      Frac = 2; {Number of digits after the decimal point in average score.}  
var Test1, Test2: integer;  
    Ave: real;  
    Ans: integer;  
  
procedure ReadScores(var Score1, Score2: integer);  
var Ans: char;  
begin{ReadScores}  
  writeln('Enter your two test scores:');  
  readln(Score1, Score2);  
  writeln('The scores are ', Score1, Score2);  
  writeln('Is that correct? (y/n)');  
  readln(Ans);  
  if Ans <> 'y' then  
    begin{then}  
      writeln('OK. Try again. Two scores:');  
      readln(Score1, Score2)  
    end {then}  
end; {ReadScores}  
  
procedure Message(Ave: real);  
begin{Message}  
  writeln('Your average is ', Ave :Width:Frac);  
  writeln('To pass you need at least ', Passing);  
  if Ave >= Passing then  
    writeln('You are passing so far.')  
  else  
    writeln('Warning: you are failing!')  
end; {Message}  
  
begin{Program}  
  writeln('Begin assessment of class work:');  
  writeln('How many tests have you taken?');  
  readln(Ans);  
  if Ans <> 2 then  
    writeln('See the instructor.')  
  else  
    begin{2 tests}  
      ReadScores(Test1, Test2);  
      Ave := (Test1 + Test2)/2;  
      Message(Ave)  
    end {2 tests}  
end. {Program}
```

**Sample Dialogue 1**

Begin assessment of class work:  
How many tests have you taken?  
1  
See the instructor.

**Sample Dialogue 2**

Begin assessment of class work:  
How many tests have you taken?  
2  
Enter your two test scores:  
57 90  
The scores are 57 90  
Is that correct? (y/n)  
n  
OK. Try again. Two scores:  
75 90  
Your average is 82.50  
To pass you need at least 70  
You are passing so far.

**Figure 5.3**  
(continued)

ask if the scores are correct; if they are not, the student is given a chance to reenter the scores. A complete breakdown of subtasks for the case of exactly two tests is shown in the data flow diagram in Figure 5.2. The various variables to be used are also shown there. The arrows indicate the data flow. If an arrow associated with some variable is pointing away from a box, and the subtask represented by that box is implemented as a procedure, then we must use a variable parameter for that piece of data. When the procedure is called, the variable will be substituted for the formal variable parameter. If the only arrow associated with a variable listed above a box points toward the box, and the subtask represented by that box is implemented as a procedure, then we will use a value parameter that takes the value of that variable.

Notice that the variable *Ans* is never used outside of the procedure *Read-Scores*; no arrows associated with the variable *Ans* point out of or into the *top* of the box for that procedure. *Ans* is used to hold an answer from the user that is only needed by that procedure and is never passed out of the procedure. Whenever a variable is used only within a procedure, it should be made a local variable; therefore, we will make *Ans* a local variable.

The complete program is given in Figure 5.3. Notice that the complete program has both a global and a local variable named *Ans*. In both cases, this is the most natural name to use. Since one of the two variables is local, this coincidence of names is not a problem.

---



## Other Local Identifiers

Identifiers other than variable names may be local to a procedure. Any kind of declaration allowed in a Pascal program is also allowed in a Pascal procedure. A procedure may have local named constants and local procedures of its own, as well as local variables. The interpretation of these other locally declared identifiers is similar to that of local variables. The local named constant or local procedure exists only while the procedure in which it is declared is executing. Outside of the procedure, you can reuse a local identifier to name something else. The ordering of local declarations within a procedure is the same as the ordering of declarations within a program.

*local  
declarations*

### Program

```

program Talk2(input, output);
const OpeningLine = 'Hello, my name is Ronald Gollum.';
      Compliment = 'You're a wonderful person.';
      Farewell = 'I hope we meet again.';

procedure BreakIce;
const OpeningLine = 'Haven't we met somewhere before?';
var FirstI, LastI: char;
procedure Compliment;
begin{Compliment}
  writeln('A lovely name. ');
  writeln('I really like that name. ')
end; {Compliment}

begin{BreakIce}
  writeln(OpeningLine);
  readln; {Discards the user's answer.}
  writeln('What is your first name? ');
  readln(FirstI);
  Compliment;
  writeln('What is your last name? ');
  readln(LastI);
  Compliment;
  writeln('Pleased to meet you ', FirstI, ', ', LastI, ' ');
end; {BreakIce}

begin{Program}
  writeln(OpeningLine);
  BreakIce;
  writeln(Compliment);
  writeln(Farewell)
end. {Program}

```

**Figure 5.4**  
Program using  
local identifiers.

### Sample Dialogue

Hello, my name is Ronald Gollum.  
 Haven't we met somewhere before?  
**I don't think so.**  
 What is your first name?  
**Jane**  
 A lovely name.  
 I really like that name.  
 What is your last name?  
**Doe**  
 A lovely name.  
 I really like that name.  
 Pleased to meet you J.D.  
 You're a wonderful person.  
 I hope we meet again.

**Figure 5.4**  
 (continued)

*local  
 constants*

*local  
 procedures*

The program in Figure 5.4 contains a procedure that has a local constant and a local procedure, in addition to local variables. There are two string constants called `OpeningLine`, one global and one local to the procedure `BreakIce`. The identifier `Compliment` is defined globally to be a string constant. Within the procedure declaration `BreakIce`, however, it is also declared to be the name of a local procedure. Within the procedure `BreakIce`, the identifier `Compliment` names a procedure. Outside of the procedure `BreakIce`, it names a string constant.

An advantage of local procedures is that they make the calling procedure a self-contained unit. Hence, if they are short and not used outside of the calling procedure, it makes sense to use local procedures. However, large procedures are seldom made local to other procedures. The inclusion of numerous long local procedures can separate a procedure heading from the main part of the procedure and so make the calling procedure awkward to read.

## Pitfall

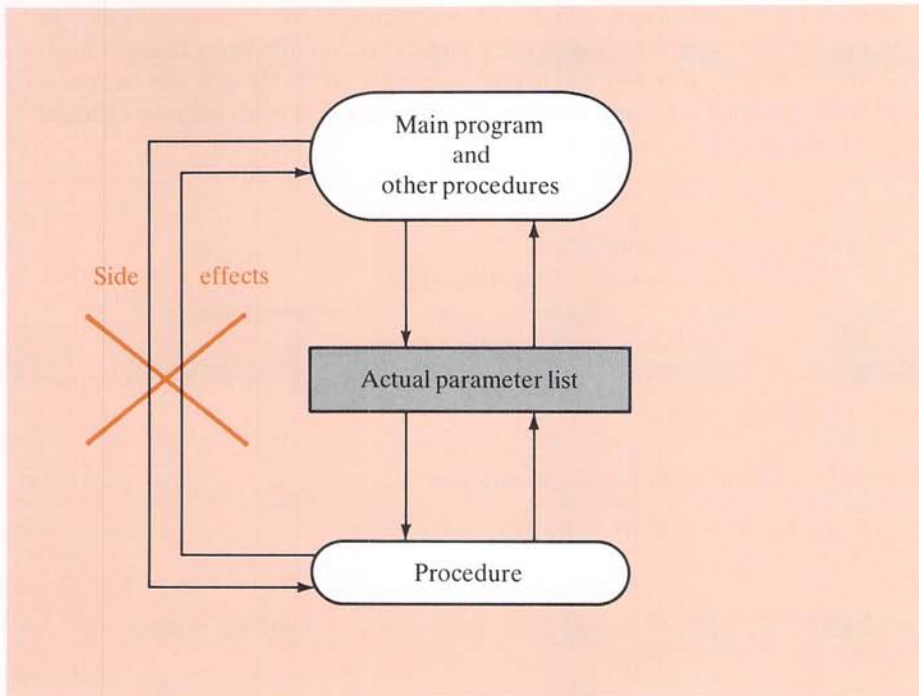
### Use of Global Variables

Pascal allows procedures to change global variables that are not actual parameters. If `X` is a global variable of type `integer`, then the compiler will accept and translate the following procedure:

```
procedure BadForm;
begin{BadForm}
  writeln('Enter a number: ');
  readln(X);
  writeln('I'll give that number to the program')
end; {BadForm}
```

However, it is almost always a bad idea to use global variables in this way.

*the problem  
 with global  
 variables*



**Figure 5.5**  
**Side effects.**

Procedures should be self-contained units that are meaningful outside the context of the program. This means that the procedure's interaction with the rest of the program should be entirely through parameters. Any other interaction between the procedure and the rest of the program is referred to as a *side effect*. As indicated in Figure 5.5, side effects are considered undesirable. If a procedure changes a global variable (other than an actual parameter), that is a side effect. Except in very rare circumstances, global variables should not appear anywhere in a procedure declaration. If a temporary variable is needed in a procedure, use a local variable, not a global variable. If you want the program to change the value of a global variable, use a formal variable parameter in the procedure declaration, and then use the global variable as the actual parameter in a procedure call.

The prohibition against global variables is often phrased as “no global variables in procedures,” which has led to confusion on the part of some novice programmers. The prohibition does not mean that procedures cannot manipulate global variables. In fact, changing global variables is the common and accepted way for one procedure to pass information to another. However, those global variables should always be passed to the procedure as actual parameters. Global variables should not be used directly in the statements within the body of a procedure declaration; instead, use a formal parameter or local variable.

Although global variables should not appear in a procedure declaration, it is perfectly acceptable to use globally defined named constants in a procedure dec-

*global  
constants*



laration (as we did in Figure 5.3). The difference is that constants cannot be changed by the procedure, and so there is no danger of their being changed inadvertently. In fact, it is very convenient to display all the defined constants in a program together at the top of the program so that they can easily be changed should the program ever need to be revised.

---

He was a local boy,  
not known outside his home town.

---

*Common saying*

## Self-Test Exercises

1. Predict the output of the following program:

```
program Exercise1(input, output);
var N: integer;

procedure Sally;
var N: integer;

begin{Sally}
  N := 7;
  writeln(N)
end; {Sally}

begin{Program}
  N := 11;
  Sally;
  writeln(N)
end. {Program}
```

2. Predict the output of the following program:

```
program Exercise2(input, output);
var Bozo: integer;

procedure Test;
const Bozo = 'Hi Folks';
begin{Test}
  writeln(Bozo, ' in procedure')
end; {Test}

begin{Program}
  Bozo := 21;
  Test;
  writeln(Bozo, ' outside of procedure')
end. {Program}
```

3. Predict the output of the following program:

```

program Exercise3(input, output);
var A, B, C: integer;

procedure Funny(var X, Y: integer);
var C: integer;
begin{Funny}
  X := 4; Y := 5; C := 6;
  writeln(X, Y, C)
end; {Funny}

begin {Program}
  A := 1; B := 2; C := 3;
  writeln(A, B, C);
  Funny(A, B);
  writeln(A, B, C)
end. {Program}

```

4. What will be the output of the program in Exercise 3 if the procedure call is changed to

```
Funny(B, A)
```

5. What will be the output of the program in Exercise 3 if the procedure call is changed to

```
Funny(A, A)
```

6. What will be the output of the program in Exercise 3 if the procedure call is changed to

```
Funny(A, C)
```

(This one is tricky. If you do not understand the answer, you may wish to leave it for now and return to it after you have had more practice with local variables.)

## Implementation of Value Parameters

Formal *value* parameters are implemented as local variables. A formal value parameter is thus a bit more than a formal blank. It is a special kind of local variable that is used to receive values when the procedure is called. When a procedure with formal value parameters is called, it must be supplied with one actual value parameter to correspond to each formal value parameter. The value of each formal value parameter is then initialized to the value of the corresponding actual parameter.

Since a formal value parameter is a local variable, you can use it just like any other local variable. Figure 5.6 illustrates a value parameter being used as a local variable. The formal value parameter `Minutes` in the procedure `OutputTime` is a local variable that is changed by the procedure. Only the value of the actual parameter `Time-Worked` is used, and so that variable is not changed by the procedure call

```
OutputTime(TimeWorked)
```

*formal value  
parameters  
used as local  
variables*

**Program**

```

program Payroll(input, output);
  {Input is the number of minutes worked. Output is the time worked in hours
  and minutes as well as the pay due at a rate of Rate cents per minute.}
  const Rate = 20; {Cents per minute.}
        MoneyLength = 7; {Field width for PayDue.}
  var TimeWorked: integer;
      PayDue: real;

  procedure OutputTime(Minutes: integer);
    {Outputs Minutes number of minutes as hours and minutes. The value
    of the actual parameter corresponding to Minutes is unchanged.}
    var Hours: integer;
    begin {OutputTime}
      Hours := Minutes div 60;
      Minutes := Minutes mod 60;
      writeln(Hours, ' hours and ', Minutes, ' minutes')
    end; {OutputTime}

  procedure ComputePay(Minutes: integer; var Pay: real);
    {Minutes is the time worked in minutes. Rate is a global constant equal to the
    pay rate expressed as pennies per minute. Pay is set to the pay due expressed in dollars.}
    var PennyPay: integer;
    begin {ComputePay}
      PennyPay := Rate * Minutes;
      Pay := PennyPay / 100
    end; {ComputePay}

  begin {Program}
    writeln('Enter the number of minutes you worked: ');
    readln(TimeWorked);
    write('You worked: ');
    OutputTime(TimeWorked);
    ComputePay(TimeWorked, PayDue);
    writeln('At ', Rate, ' cents per minute,');
    writeln('you earned: $', PayDue : MoneyLength: 2)
  end. {Program}

```

**Sample Dialogue**

```

Enter the number of minutes you worked:
62
You worked:    1 hours and    2 minutes
At    20 cents per minute,
you earned: $    12.40

```

**Figure 5.6**

A value parameter  
used as a local  
variable.



## Pitfall

### Inadvertent Local Variables

If you want a procedure to change the value of a variable, then you must use an actual variable parameter. This means that the corresponding formal parameter must be preceded by *var*. If you carelessly omit the *var*, the procedure will have a value parameter where you meant to have a variable parameter. This can be very frustrating because it will undoubtedly result in incorrect output. Yet the program will run with no error messages and is likely to look correct, since the only mistake is typographically very minor. If you make such a mistake and attempt to locate it by tracing variables, you will find that the procedure does not change the value of the actual parameter. This is because a formal value parameter is a local variable—if its value is changed in the procedure, then, as with any local variable, that change will be lost when the procedure call is completed. Any time you find a procedure that does not change a parameter value when it should, check for a missing *var*.

For example, the procedure `ReadScores` in Figure 5.7 is supposed to use variable parameters. Since the *var* was omitted, the parameters became value parameters, and so the procedure has no effect on the value of the actual parameters.

---

## Scope of an Identifier

A variable that is declared in a procedure is local to that procedure. Its meaning is confined to that procedure and you need not even be aware of its existence at any time other than when you write the procedure. That is the beauty of local declarations. As long as you think of each procedure as a self-contained unit and design each unit separately, you will never even notice that the same identifier names two or more different variables in a program. Life is not as easy for the computer. The computer must decide the meaning of each variable or other identifier in the program. If there is more than one declaration for a single variable name, it must decide which declaration goes with each of the occurrences of that variable name. How it makes this decision is the topic of this section.

A set of declarations, possibly prefaced by a parameter list, together with the body of statements to which they apply, is a program unit called a *block*. This term is used to refer to a procedure and also to an entire program. The syntax for a block is summarized in Figure 5.8. Notice that a procedure is thus divided into two overlapping parts: a procedure heading and a block; the formal parameter list is considered to be both part of the heading and part of the block.

A procedure declaration consists of the word *procedure*, followed by the procedure name, followed by a block and a semicolon. All the identifiers described at the

*block*

*scope*

start of a block are said to be *local to that block* or, equivalently, to have that block as their *scope*. The meaning given to an identifier by the parameter list or by a declaration applies only within the block.

If an identifier is declared or is listed as a formal parameter in each of two blocks, one within the other, then its meaning *within the inner block* is the one *determined by the declaration or parameter list at the start of the inner block*.

**Program**

```

program Careless(input, output);
{Illustrates the effect of omitting the var from the parameter list of
the procedure ReadScores of Figure 5.3.}
var S1, S2: integer;

procedure ReadScores(Score1, Score2: integer);
var Ans: char;
begin{ReadScores}
  writeln('Enter your two test scores:');
  readln(Score1, Score2);

  writeln('The scores are ', Score1, Score2);
  writeln('Is that correct? (y/n)');
  readln(Ans);

  if Ans <> 'y' then
    begin{then}
      writeln('OK. Try again. Two scores:');
      readln(Score1, Score2)
    end {then}
end; {ReadScores}

begin{Program}
  S1 := 1; S2 := 2;
  ReadScores(S1, S2);
  writeln('Procedure call is over. ');
  writeln('S1 = ', S1, ' S2 = ', S2)
end. {Program}

```

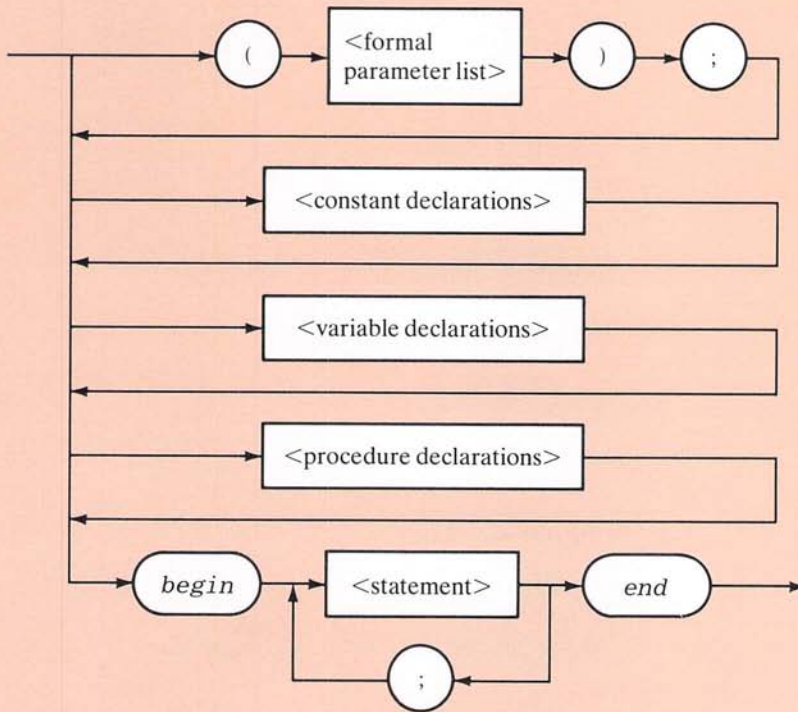
**Sample Dialogue**

```

Enter your two test scores:
98 100
The scores are 98 100
Is that correct? (y/n)
y
Procedure call is over.
S1 =   1 S2 =   2

```

**Figure 5.7**  
Effect of forgetting  
a var.



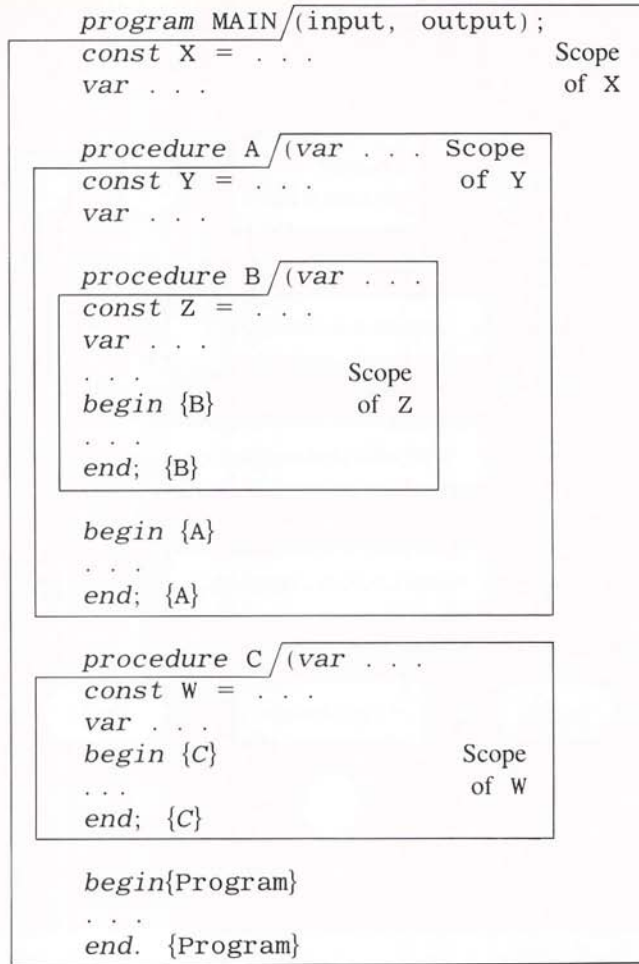
**Figure 5.8**  
Syntax of a block.

If an identifier is given a meaning within the block of a procedure, then when the procedure is called, the identifier takes on that meaning, and when the procedure call is over, it loses that identity. If the same identifier had a meaning before the procedure call, then that meaning is suspended for the duration of the procedure call but returns with all its properties preserved when the procedure call is completed. Figure 5.9 shows a possible arrangement of blocks for a program.

Figure 5.10 contains a program designed to illustrate the notion of scope. The single identifier *X* appears in three different blocks so that the program contains three different variables named *X*. Each occurrence of the identifier *X* names one of these three variables. The three different shadings indicate the three areas of meaning for the identifier *X*.

When a procedure with a local variable, say *X*, is called, the computer saves the value of any global variable called *X* and then executes the procedure call. When the procedure call is completed, the global variable *X*, with its saved value, is restored. If





**Figure 5.9**  
Scope of  
identifiers.

there is a procedure call within a procedure and both procedures have a local variable called X, then there can be two saved values for the identifier X. This is illustrated in Figure 5.11. The procedure calls in the figure are expanded to show the entire procedure and thereby display all the statements of all the procedures in the order in which they are actually executed. When the procedure ProB is called, the value of the global variable X is saved, and the identifier X then names the local variable in ProB. When the procedure ProA is called, the value of the local variable X in ProB is saved, so that two values are now saved, and the identifier X then names the local variable in the procedure ProA. When the execution of the procedure ProA is completed, the local variable in ProB is restored. When the execution of the procedure ProB is completed, the global variable X is restored. In more complicated programs, even more values may need to be saved.

**Program**

```

program ShowScope(input, output);
var X: integer;

procedure ProA;
var X: integer;
begin{ProA}
  writeln('****Start ProA');
  X := 3;
  writeln('****In ProA, X = ', X);
  writeln('****End ProA')
end;{ProA}

procedure ProB;
var X: integer;
begin{ProB}
  writeln('*Start ProB');
  X := 2;
  writeln('*In ProB, X = ', X);
  ProA;
  writeln('*In ProB, X = ', X);
  writeln('*End ProB')
end;{ProB}

begin{Program}
  writeln('Start Program');
  X := 1;
  writeln('Global X = ', X);
  ProB;
  writeln('Global X = ', X);
  writeln('End Program')
end. {Program}

```

**Output**

```

Start Program
Global X = 1
*Start ProB
*In ProB, X = 2
****Start ProA
****In ProA, X = 3
****End ProA
*In ProB, X = 2
*End ProB
Global X = 1
End Program

```

**Figure 5.10**  
One identifier in  
three scopes.

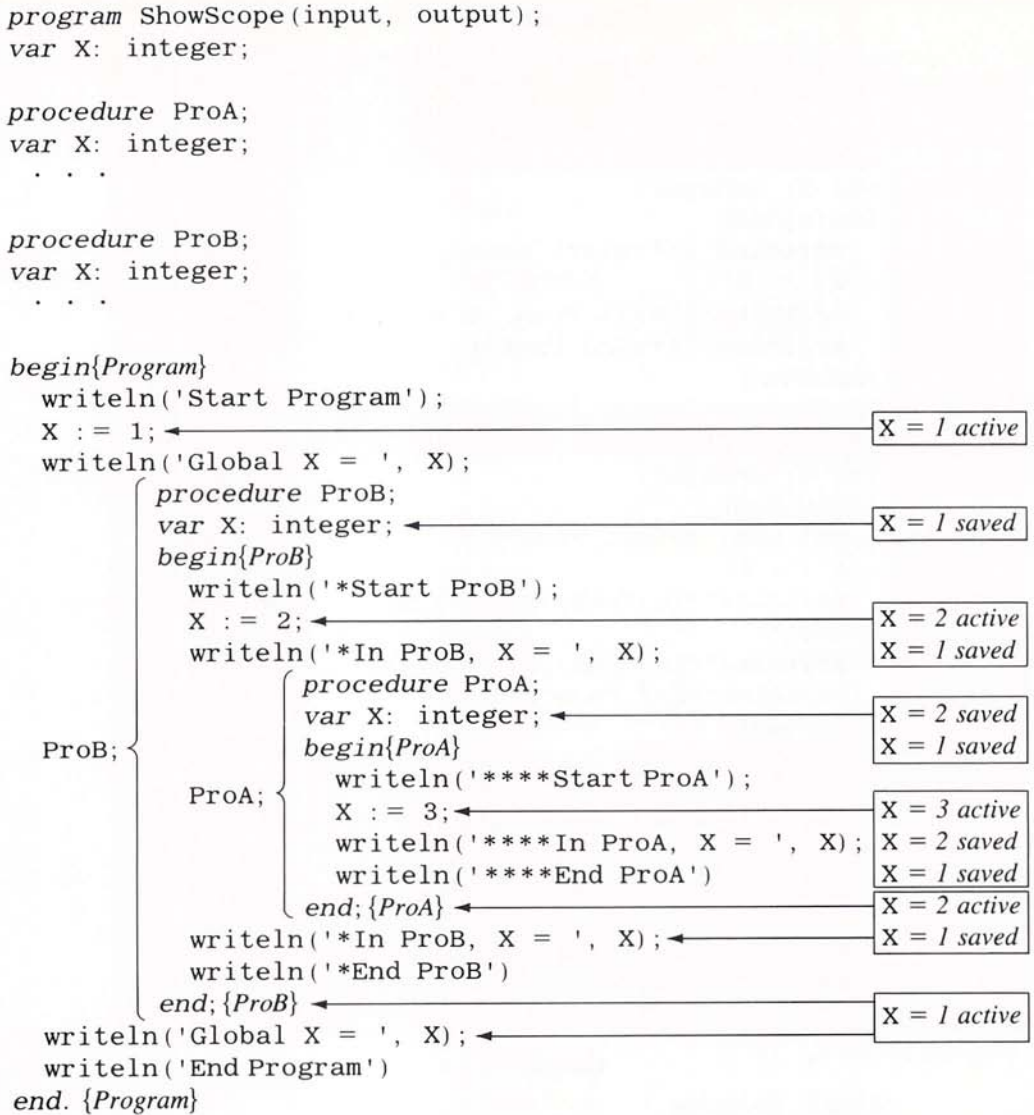


Figure 5.11  
Explanation of  
Figure 5.10.

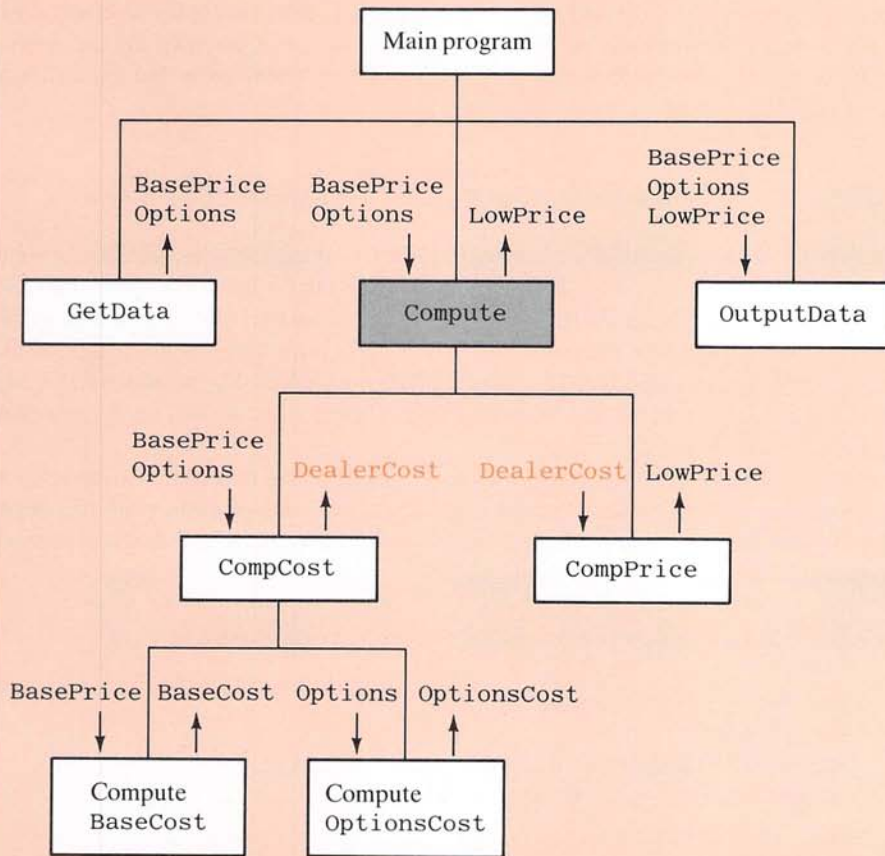
## Case Study

### Automobile Bargaining

#### Problem Definition

A consumer service organization has commissioned you to write a program that will help people bargain for a low price when purchasing an automobile. The program will





### Data Summary

BasePrice: list price of the basic automobile.

Options: list price of desired options.

BaseCost: cost to dealer of basic automobile, including any delivery charge.

OptionsCost: cost to dealer of options.

DealerCost: cost to dealer of the automobile with options.

LowPrice: lowest price acceptable to dealer.

**Figure 5.12**  
Data flow diagram  
for automobile  
pricing.

take as input the list price of the basic automobile and the list price of the desired options and will then output the lowest price that a dealer will accept. The buyer can then hold out for this price. The organization tells you that dealers will accept a price that is 10% above the cost to the dealer of the automobile plus options. It also tells you that the cost of the automobile or options is one-half the list price, except for economy cars, on which the dealer must pay an additional delivery charge. Currently, all cars with a list price of less than \$12,000 are considered to be economy cars, and the delivery charge is \$300.

### Discussion

This problem breaks down into three main subtasks: input the data, perform a computation, and output the results. The computation subtask further decomposes into two smaller subtasks: determine the dealer's cost, and calculate the lowest acceptable price. Since the dealer's cost for the basic automobile is slightly different from that of the options, we can subdivide this task into two subcomputations. The breakdown of tasks into subtasks is shown in the data flow diagram in Figure 5.12. We will implement each subtask as a procedure.

*local  
variable*

The variable `DealerCost` is never used outside of the procedure `Compute`. It is used to hold an intermediate value that is calculated by the procedure but that never is passed out of the procedure. Hence, we will make it a local variable in the procedure `Compute` so the procedure will read

```
procedure Compute(BasePrice, Options: real;
                  var LowPrice: real);
var DealerCost: real;
begin{Compute}
  CompCost(BasePrice, Options, DealerCost);
  CompPrice(DealerCost, LowPrice)
end; {Compute}
```

### ALGORITHM

The algorithm for computing the dealer's cost is obtained by formalizing the description given to us in the problem definition.

```
if the car is an economy car then
  BaseCost := 0.5 * (the base price) + (the delivery charge)
else {it is not an economy car}
  BaseCost := 0.5 * (the base price);
OptionsCost := 0.5 * (the list price of all options);
(the dealer's cost) := BaseCost + OptionsCost
```

The algorithm is implemented as the procedure `CompCost`. The variables `BaseCost` and `OptionsCost` are not used outside of the procedure `CompCost`, and they can therefore be local variables. The complete program is given in Figure 5.13.

---

## Testing Procedures

Because they divide big tasks into smaller tasks of more manageable size, procedures make programs easier to write and easier to change. You can solve each subtask separately, write it up as a procedure, and then test it separately.

As a sample case, consider the automobile-pricing program in Figure 5.13. We can test each of the procedures separately. We can check the procedure `GetData` by using a program such as the one in Figure 5.14. Having tested this procedure, we can go on to test the other procedures.

We can test the procedure `CompCost` with a program such as the one in Figure 5.15. The procedure `CompPrice` can be tested by a similar program. Once these two procedures have been tested, we can then test the procedure `Compute`. Testing programs like those shown in Figures 5.14 and 5.15 are often called *driver programs*.

Each procedure in a program should be tested separately. The method we outlined for testing the automobile-pricing program is called *bottom-up testing* and is one of two basic methods for testing a program. In the bottom-up testing strategy, each procedure is tested and debugged before any procedure that uses it is tested. One possible order for bottom-up testing of our automobile-pricing program is given in Figure 5.16, on page 169.

If you test each procedure separately, you will find most of the mistakes in your program. Moreover, you will find out which procedure contains the mistake. If instead you just test the entire program, and if there is a mistake, then you will probably find out that there is a mistake, but you may have no idea of where it is. Even worse, you may think you know where the error is but be mistaken.

Testing each procedure separately may sound like a very time-consuming process. However, if you follow this strategy, you will find that the time saved by quickly locating bugs will allow you to write the final program faster than if you wrote and tested the program as a single undivided unit.

*driver  
programs*

*bottom-up  
testing*

---

## Top-Down and Bottom-Up Strategies

The bottom-up testing strategy, presented in the last section, is a reasonable approach to testing a small program or small portions of a larger program. However, when testing a large program, the bottom-up strategy does not always make sense. The best way to design a program is top-down. First, break the task into subtasks. Then write procedures for the subtasks. These procedures will contain calls to yet other procedures to perform smaller subtasks. In order to test the basic design strategy, it frequently is a good idea to test each procedure before going on to design the procedures it uses. This method of testing is called *top-down testing*. For example, a possible top-down order for testing the procedures in our automobile-pricing program is given in Figure 5.17.

How can you test a procedure or program, such as `InsideScoop`, before writing the procedures it uses? The answer is to write simple versions of the missing procedures and to use these simplified versions to test the calling program (or procedure).

*top-down  
testing*

---



### Program

```

program InsideScoop(input, output);
{Determines a dealer's minimum acceptable price on an automobile.}
const Markup = 0.10; {10% over cost.}
      LuxuryPrice = 12000; {Autos below this price are economy class.}
      Delivery = 300; {Delivery charge on nonluxury models.}
      WholesaleFactor = 0.5;
      {Wholesale is 50% of list price.}
      Width = 8; {Field width for price of auto.}
var BasePrice, Options, LowPrice: real;

procedure GetData(var BasePrice, Options: real);
{Reads list BasePrice (without options) and list price of Options.}
begin{GetData}
  writeln('Enter the base sticker price (without options):');
  readln(BasePrice);
  writeln('Enter the total sticker price for all options:');
  readln(Options)
end; {GetData}

procedure CompCost(BasePrice, Options: real;
                  var DealerCost: real);
{Calculates DealerCost from list prices for BasePrice and Options.}
var BaseCost, OptionsCost: real;
begin{CompCost}
  if BasePrice < LuxuryPrice then
    BaseCost := WholesaleFactor * BasePrice + Delivery
  else
    BaseCost := WholesaleFactor * BasePrice;
    OptionsCost := WholesaleFactor * Options;
    DealerCost := BaseCost + OptionsCost
end; {CompCost}

procedure CompPrice(DealerCost: real; var LowPrice: real);
{Is given DealerCost = the total cost of the automobile to the dealer.
Sets LowPrice equal to the lowest price that dealers will accept.}
begin{CompPrice}
  LowPrice := (1 + Markup) * DealerCost
end; {CompPrice}

procedure Compute(BasePrice, Options: real; var LowPrice: real);
{Is given the list price for BasePrice and Options.
Sets LowPrice equal to the lowest price that dealers will accept.}
var DealerCost: real;

```

Figure 5.13

Automobile-pricing  
program.

```

begin{Compute}
  CompCost(BasePrice, Options, DealerCost);
  CompPrice(DealerCost, LowPrice)
end; {Compute}

procedure OutputData(BasePrice, Options, LowPrice: real);
begin{OutputData}
  writeln('List price for basic model: $', BasePrice :Width:2);
  writeln('List price for options: $', Options :Width:2);
  writeln('Dealer''s lowest total price: $', LowPrice :Width:2);
  writeln('Go for it!')
end; {OutputData}

begin{Program}
  writeln('I will help you get a bargain on your new car. ');
  GetData(BasePrice, Options);
  Compute(BasePrice, Options, LowPrice);
  OutputData(BasePrice, Options, LowPrice)
end. {Program}

```

#### Sample Dialogue

I will help you get a bargain on your new car.  
 Enter the base sticker price (without options):  
**13000**  
 Enter the total sticker price for all options:  
**4000**  
 List price for basic model: \$13000.00  
 List price for options: \$ 4000.00  
 Dealer's lowest total price: \$ 9350.00  
 Go for it!

**Figure 5.13**  
(continued)

```

program Test1(input, output);
const Width = 8;
var BasePrice, Options: real;

procedure GetData(var BasePrice, Options: real);
{Reads list BasePrice (without options) and list price of Options.}
begin{GetData}
  writeln('Enter the base sticker price (without options): ');
  readln(BasePrice);
  writeln('Enter the total sticker price for all options: ');
  readln(Options)
end; {GetData}

begin{Program}
  GetData(BasePrice, Options);
  writeln('BasePrice = ', BasePrice :Width:2);
  writeln('Options price = ', Options :Width:2)
end. {Program}

```

**Figure 5.14**  
Test 1.

### Program

```

program Test2(input, output);
const LuxuryPrice = 12000; {Autos below this price are economy class.}
      Delivery = 300; {Delivery charge on nonluxury models.}
      WholesaleFactor = 0.5; {Wholesale is 50% of list price.}
      Width = 8; {Field width for price of auto.}
var BasePrice, Options, DealerCost: real;

procedure CompCost(BasePrice, Options: real;
                  var DealerCost: real);
{Calculates DealerCost from list prices for BasePrice and Options.}
var BaseCost, OptionsCost: real;
begin{CompCost}
  if BasePrice < LuxuryPrice then
    BaseCost := WholesaleFactor * BasePrice + Delivery
  else
    BaseCost := WholesaleFactor * BasePrice;
  OptionsCost := WholesaleFactor * Options;
  DealerCost := BaseCost + OptionsCost
end; {CompCost}

begin{Program}
  writeln('Enter list price for BasePrice and Options:');
  readln(BasePrice, Options);
  CompCost(BasePrice, Options, DealerCost);
  writeln('BasePrice = ', BasePrice :Width:2);
  writeln('Options price = ', Options :Width:2);
  writeln('DealerCost = ', DealerCost :Width:2)
end. {Program}

```

### Sample Dialogue

```

Enter list price for BasePrice and Options:
13000 4000
BasePrice =      13000.00
Options price =    4000.00
DealerCost =      8500.00

```

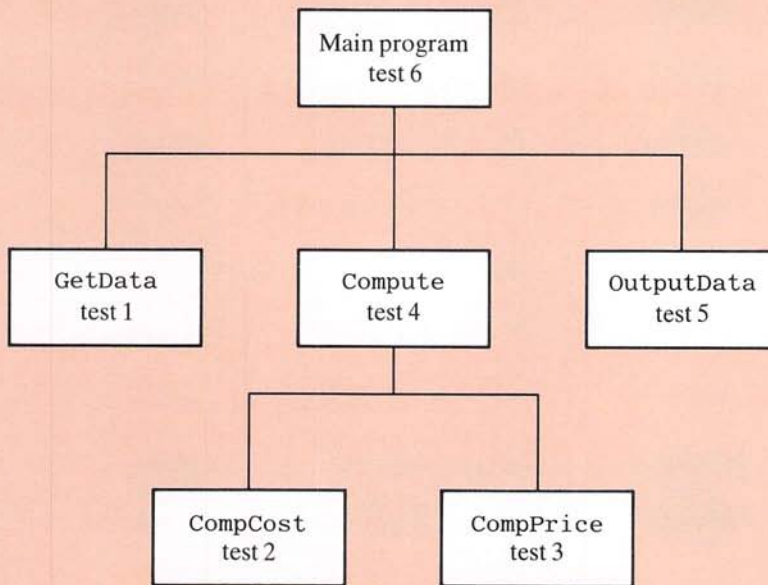
**Figure 5.15**  
**Test 2.**

*stub  
programs*

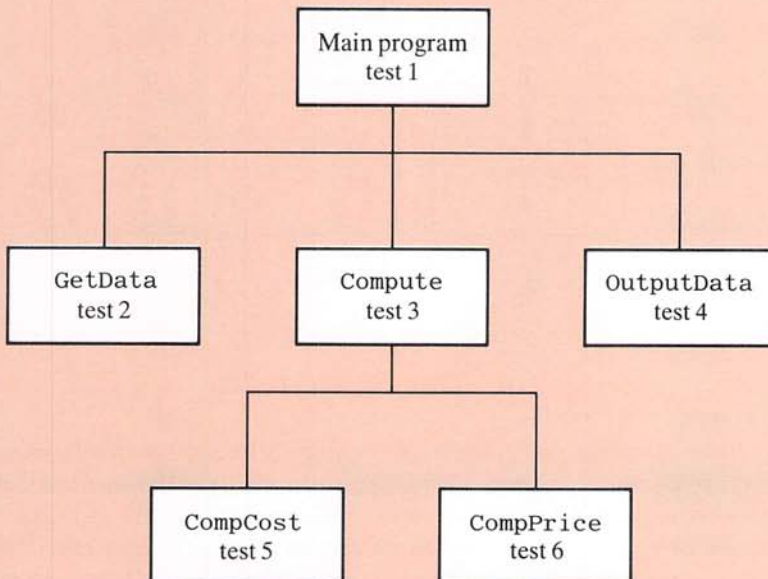
The simple version will not do what the final procedure is supposed to do, but it will behave like the procedure is supposed to behave on the test cases for which it is used. For example, a temporary version of Compute is shown in Figure 5.18. Early versions of the program with such simplified versions of some procedures are frequently called *stub programs*.

The advantages of bottom-up testing are obvious. Each procedure is tested with fully debugged, final versions of the procedures it uses. Hence, any problems discovered can be attributed to the procedure being tested. The advantage of top-down testing





**Figure 5.16**  
One order for  
bottom-up testing.



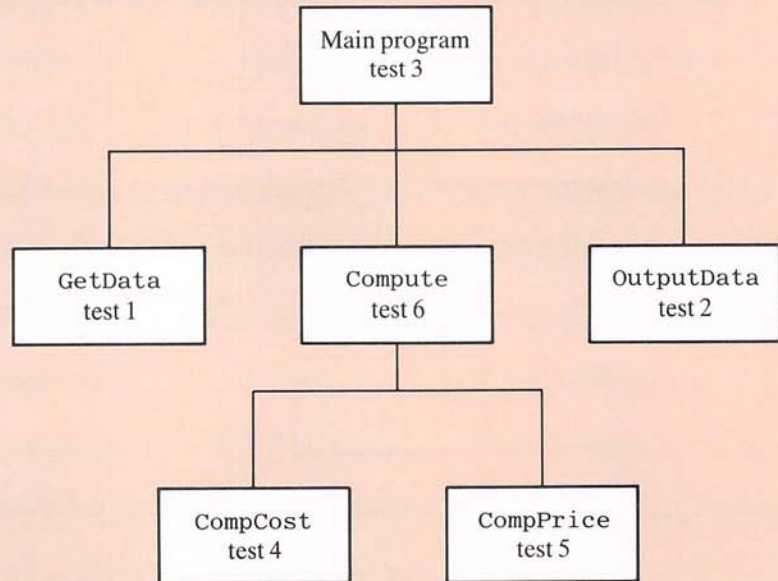
**Figure 5.17**  
One order for top-  
down testing.

```

procedure Compute(BasePrice, Options: real; var LowPrice: real);
{Is given the list price for BasePrice and Options.
Sets LowPrice equal to the lowest price that dealers will accept.}
begin{Compute}
  writeln('BasePrice = ', BasePrice :8:2);
  writeln('Options price = ', Options :8:2);
  writeln('Enter LowPrice: ');
  readln(LowPrice)
end; {Compute}

```

**Figure 5.18**  
Temporary version  
of procedure  
Compute.



**Figure 5.19**  
One order for a  
mixed strategy.

is that it allows you to test your basic design strategy before you get too far along in the design process. It allows you to test whether a particular breakdown of tasks into subtasks will work as desired. If it will not, there is no need to solve the subtasks; instead, you need to back up and rethink the way you divided the task into subtasks. It also lets you test the exact specification of what each procedure should do before you go ahead and design the details of how it will do it.

Sometimes it is best to test bottom-up, sometimes it is best to test top-down and sometimes it is best to mix the two strategies, testing some procedures bottom-up and some top-down. Figure 5.19 gives a mixed strategy that we could have used to design and test our automobile-pricing program. If the procedures are designed in the order given there, then the simplified version of `Compute` from Figure 5.18 can be used until the final version of the procedure is written.

*mixed  
strategies*

## Preconditions and Postconditions

One way to document what a procedure does is by means of special-purpose assertion comments known as *preconditions* and *postconditions*. A precondition is an assertion that states what is expected to be true whenever the procedure is used. The procedure should not be used and cannot be expected to perform correctly unless the precondition holds. A postcondition states the relevant facts that will be true after the procedure is executed in a situation in which all the preconditions hold. For example, the following are possible pre- and postconditions for the procedure `Compute` in Figure 5.13:

```
procedure Compute(BasePrice, Options: real; var LowPrice: real);
{Precondition: BasePrice = list price of basic automobile,
  Options = list price of options;
 Postcondition: LowPrice = dealer's lowest acceptable price.}
```

The case study in the next section provides an additional illustration of pre- and postconditions.

You should include a comment with each procedure heading. The comment need not follow this exact format of writing preconditions and postconditions, but it should describe any conditions that must hold before the procedure is called and should say what effects will be produced by executing the procedure. The comment should also explain everything else that a programmer using the procedure needs to know, such as what other procedures it calls, what defined constants it uses, and what variables it changes. One popular method for displaying this comment is to frame the comment in a box to make it stand out. For instance, the following would be a good longer comment to add to the procedure `Compute` discussed in the previous paragraph.

```
procedure Compute(BasePrice, Options: real; var LowPrice: real);
{ *****
 *Precondition: BasePrice = list price of basic automobile,
 *Options = list price of options;
 *Postcondition: LowPrice = dealer's lowest acceptable price.
 *Calls the procedures CompCost and CompPrice.
 *Uses the defined constants LuxuryPrice, WholesaleFactor, Delivery, and Markup.
 * ***** }
```

On large programming projects involving more than one programmer, this comment should also contain the author's name and the date written, as well as the date and na-



ture of any modifications to the procedure. In this book we will use a somewhat shorter comment format in order to save space.

---

## Case Study

---

### Calculating Leap Years

#### Problem Definition

We want to write a program that takes a year as input and provides output that tells whether or not that year is a leap year.

#### Discussion

#### SUBALGORITHMS

For most years it is very easy to decide whether the year is a leap year. If the year is divisible by 4, then it is a leap year. So 2004 is a leap year, but 2001 is not. This rule is based on the fact that it takes 365 and one-quarter days for the earth to revolve around the sun. Every fourth year, we add up the four extra quarter-days and insert an extra full day in February to make the calendar consistent with the earth's movement. Unfortunately for calendar makers, the earth actually takes a bit less than 365.25 days to complete its trip around the sun, and so leap years should add a bit less than one full day. This is not practical, however, and so a full day is added and then after a few centuries a leap year day is skipped, which effectively subtracts a day and makes things balance out. The exact rule says that if a year is divisible by 100, then it is not a leap year unless it is also divisible by 400. Therefore, although 2200 is divisible by 4, it will not be a leap year. Our algorithm will include subalgorithms for two separate calculations, one for most years and the other for years divisible by 100.

RegularCalc: {Precondition: the year is not divisible by 100.}  
 if the year is divisible by 4, then it is a leap year; otherwise it is not.  
 CenturyCalc: {Precondition: the year is divisible by 100.}  
 if the year is divisible by 400, then it is a leap year; otherwise it is not.

We will implement both of these subalgorithms as procedures with a parameter *Y* for the year. Since the year is not changed, these parameters will be value parameters. The complete program is given in Figure 5.20.

#### Program

**Figure 5.20**  
**Leap year**  
**program.**

```
program LeapYear(input, output);
{Determines whether the input year is a leap year.}
var Year: integer;
```

```
procedure RegularCalc (Y: integer);  
{Precondition: Y is not divisible by 100.  
Postcondition: Displays a message saying whether or not Y is a leap year.}  
begin{RegularCalc}  
    if (Y mod 4) = 0 then  
        writeln(Y, ' is a leap year!')  
    else  
        writeln(Y, ' is not a leap year.')  
    end; {RegularCalc}  
  
procedure CenturyCalc (Y: integer);  
{Precondition: Y is divisible by 100.  
Postcondition: Displays a message saying whether or not Y is a leap year.}  
begin{CenturyCalc}  
    if (Y mod 400) = 0 then  
        writeln(Y, ' is a leap year!')  
    else  
        writeln(Y, ' is not a leap year.')  
    end; {CenturyCalc}  
  
begin{Program}  
    writeln('Enter a year. ');  
    readln(Year);  
    if (Year mod 100) = 0 then  
        CenturyCalc (Year)  
    else  
        RegularCalc (Year)  
    end. {Program}
```

#### Sample Dialogue 1

```
Enter a year.  
1900  
    1900 is not a leap year.
```

#### Sample Dialogue 2

```
Enter a year.  
2000  
    2000 is a leap year!
```

#### Sample Dialogue 3

```
Enter a year.  
2004  
    2004 is a leap year!
```

**Figure 5.20**  
(continued)

---

Good things come in small packages.

*Proverb*

---

---

## Summary of Problem Solving and Programming Techniques

- Use local variables to hold any temporary information that a procedure may need for its calculations but that is not needed by any other part of the program.
- The interaction of a procedure with the rest of the program should be via parameters. Global variables normally should not appear in a procedure declaration, but they can be passed as actual parameters to the procedure.
- A formal value parameter is a local variable that is initialized to the value of the corresponding actual parameter when the procedure is called. It is possible to use it just like any other local variable.
- If a procedure does not change the value of an actual variable parameter when it should, suspect a missing *var* in the formal parameter list.
- Each procedure should be tested separately in a program that contains no procedures except the one being tested and possibly some other procedures that have already been fully tested and debugged.
- You can test a procedure before all the procedures it uses have been written. To do so, use simplified versions of the missing procedures.
- Preconditions and postconditions are a type of assertion that can be used effectively to document procedures.

---

## Summary of Pascal Constructs

### procedure declaration

Syntax:

```
procedure <procedure name> (<formal parameter list>) ;
    <local constant declarations>
    <local variable declarations>
    <local procedure declarations>
begin
    <statement 1>;
    <statement 2>;
    .
    .
    .
    <statement n>
end;
```

The statements may be any Pascal statements. See the following entries for details on declarations.

---



### local variable

A local variable is one that is declared within a procedure declaration. A local variable exists only for the duration of the procedure call. The identifier used to name a local variable can also be used to name another variable (or constant or other object) outside of the procedure declaration.

### local identifier

A local identifier is one that is declared within a procedure. The object it names exists only for the duration of the procedure call. A local identifier can also be used outside of the procedure declaration to name something else. Local variable names, local constant names, and local procedure names are examples of local identifiers.

### global variable

A variable that is declared for the entire program; that is, one whose scope is the block of the entire program.

### block

A block consists of a parameter list followed by a set of declarations, followed by the statements to which they apply, enclosed in a *begin/end* pair. (If there is no parameter list or no set of declarations, it is still a block.) For example, if you remove the procedure name and final semicolon from a procedure declaration, what is left is a block.

### scope of an identifier

The block in which an identifier is declared is called its *scope*. If an identifier is declared in two blocks, one inside the other, then its meaning in the inner block is the one declared in the inner block.

---

## Exercises

### Self-Test Exercises

7. What will be the output of the program in Exercise 3 if the procedure heading is changed to the following? (The *var* is left out.)

```
procedure Funny(X, Y: integer);
```

8. What is the output of the following program?

```
program Exercise8(input, output);  
var X, Y: char;  
  
procedure Mixed(X: char; var Y: char);  
begin{Mixed}  
  X := 'A'; Y := 'B';  
  writeln(X, Y)  
end; {Mixed}
```

```
begin{Program}  
  X := 'X'; Y := 'Y';  
  writeln(X, Y);  
  Mixed(X, Y);  
  writeln(X, Y)  
end. {Program}
```

### Interactive Exercises

9. Run the program in Figure 5.1 twice, once as shown and once omitting the local variable declaration

```
var Temp: integer;
```

10. Write a procedure to interchange the value of two variables of type `char`.

### Programming Exercises

11. Write a procedure with two variable parameters `N1` and `N2` of type `integer` that sorts the two values so that after the procedure is called, `N1` is less than or equal to `N2`. The procedure either leaves the values of the variables unchanged or else interchanges the two values. Your procedure will include a call to the procedure `Exchange` from Figure 5.1. Embed the procedure in a test program.

12. Write a program that computes the annual after-tax cost of a new house for the first year of ownership. The cost is computed as the annual mortgage cost minus the tax savings. The input should be the price of the house and the down payment. The annual mortgage cost can be estimated as 3% of the initial loan balance (credited toward paying off the loan principal) plus 10% of the initial loan balance in interest. The initial loan balance is the price minus the down payment. Assume a 35% marginal tax rate and assume that interest payments are tax deductible. The tax savings is therefore 35% of the interest payment. Use at least three procedures.

13. Write a program for the discount installment loan algorithm described in Exercise 10 of Chapter 1. Implement subtasks as procedures.

14. Write a program that reads in a length in feet and inches and then outputs the equivalent length in meters and centimeters. Use at least three procedures: one for input, one or more for calculating, and one for output. There are 0.3048 meters in a foot, 100 centimeters in a meter, and 12 inches in a foot.

15. Write a program like that of the previous exercise that converts measurements from meters and centimeters into feet and inches.

16. Write a program that reads in a weight in pounds and ounces and then outputs the equivalent weight in kilograms and grams. Use at least three procedures: one for input, one or more for calculating, and one for output. There are 2.2046 pounds in a kilogram, 1000 grams in a kilogram, and 16 ounces in a pound.

17. Write a program like that of the previous exercise that converts weights from kilograms and grams into pounds and ounces.

18. Combine, modify, and extend your programs from the previous four exercises into

---

a single program that can perform a choice of conversions from the system of weights and measures commonly used in the United States to the metric system and vice versa. It first asks the user whether he or she wants to convert from metric to U.S. measurements or from U.S. measurements to metric. It then asks the user whether he or she wishes to deal with weights or lengths. It then performs the desired calculation. For example, in one case the program asks for a weight in pounds and ounces and outputs the weight expressed in kilograms and grams. Use procedure calls as the substatements in *if-then-else* statements. Some of these procedures will in turn have *if-then-else* statements with other procedure calls for their substatements.

19. Enhance the program in Figure 4.13 so that it accepts as input an amount of dollars and cents as a value of type `real` and then outputs the numbers of the various bills and coins that equal that amount. Use bill denominations of \$20, \$5, and \$1. Also include nickels as a possible coin in this version. Use the procedure `ComputeCoins` from Figure 4.14 and add a similar procedure to compute numbers of bills.

20. Write a program that is a major improvement on the program `CashRegister` in Figure 4.5. This improved version asks for the amount tendered by the customer as well as the price and number of the items. It responds with the price, the number of items, the total bill, the amount tendered, and the change due. It then goes on to tell the cashier exactly what combination of bills and coins will equal the amount of change. Use procedures for subtasks. The program `Change2` in Figure 4.13 can be used as a model for the last task. It would pay to do the previous exercise before doing this one.

21. Write a procedure with two value parameters of type `integer` called `Number` and `Width`. The procedure writes `Number` to the screen in the conventional way, with a comma inserted if the number is 1,000 or larger. You may assume that the number is at most five digits long. `Width` is used as the field width specification for the total number of spaces *including a comma*, if there is one. Note that you will have to use other field width specifications derived from `Width` inside `write` statements within the procedure body.

22. Redo (or do for the first time) Exercise 20 from Chapter 3. Use three procedures: one for input, one for output, and one to perform the calculation.

23. Redo (or do for the first time) Exercise 26 from Chapter 3. Use three procedures: one for input, one for output, and one to perform the calculation.

24. Redo (or do for the first time) Exercise 27 from Chapter 3. Use three procedures: one for input, one for output, and one to perform the calculation.

25. Redo (or do for the first time) Exercise 29 from Chapter 3. Use three procedures: one for input, one for output, and one to perform the calculation.

26. Redo (or do for the first time) Exercise 30 from Chapter 3. Use three procedures: one for input, one for output, and one to perform the calculation.

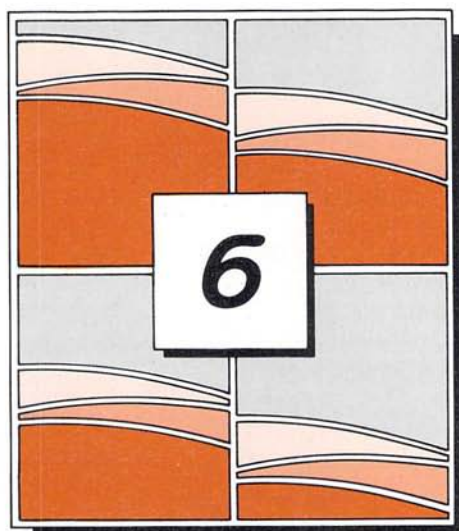
27. Redo (or do for the first time) Exercise 32 from Chapter 3. Use three procedures: one for input, one for output, and one to perform the calculation.

28. Redo (or do for the first time) Exercise 33 from Chapter 3. Use three procedures: one for input, one for output, and one to perform the calculation.

---







## ***Designing Programs That Make Choices***

"If you think we're wax-works," he said, "you ought to pay, you know. Waxworks weren't made to be looked at for nothing. Nohow!"

"Contrariwise," added the one marked "DEE," "if you think we're alive, you ought to speak."

*Lewis Carroll, Through the Looking-Glass*

## Chapter Contents

Nested Statements  
Nesting If-Then and If-Then-Else Statements  
Complex Boolean Expressions  
George Boole (Optional)  
Evaluating Boolean Expressions  
Pitfall—Undefined Boolean Expressions  
Self-Test Exercises  
Programming with Boolean Variables  
Pitfall—Omitting Parentheses in Boolean Expressions  
Boolean Input and Output

Case Study—Designing Output  
Boolean Constants and Debugging Switches  
The Case Statement  
TURBO Pascal—The Case Statement  
The Empty Statement  
Programming Multiple Alternatives  
Case Study—State Income Tax  
Case Study—Scoring Blackjack  
Summary of Problem Solving and Programming Techniques  
Summary of Pascal Constructs  
Exercises

Any programming construct that chooses one out of a number of alternative actions is called a *branching mechanism*. The *if-then* and *if-then-else* statements are two examples of branching mechanisms. In this chapter we complete our description of these two types of statements by describing the full class of boolean expressions that can be used to control how they branch. We then go on to describe boolean variables and how they are used as a programming tool. Pascal has one additional branching statement, called the *case* statement, which we introduce in this chapter. We also explore problem solving and programming techniques for designing branches in particular and programs in general. We begin with a discussion of how Pascal branching statements may be nested within other branching statements to produce complex program instructions.



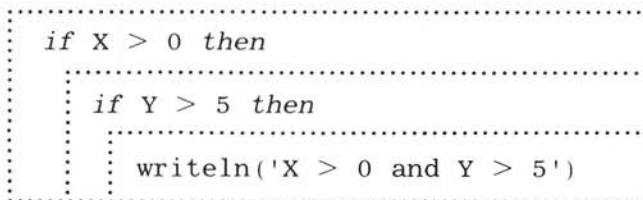
## Nested Statements

An *if-then-else* or *if-then* statement contains a smaller statement within it. Thus far we have used compound statements as well as simpler statements such as assignment statements as the substatement. In fact, any statement at all can be used. In particular, we can use an *if-then* statement within a larger *if-then* statement.

This nesting of statements inside of statements points to one peculiarity in the definitions of compound and *if-then* statements. The definitions say that they can contain any sort of statement, including one of the same kind, as illustrated in Figure 6.1. This way of defining statements may seem circular. That is because it is. Circular definitions and even circular program instructions are common in computer science, although in computer science, circular definitions and instructions are usually referred to by the word *recursive* rather than the word *circular*.

It is not always wrong or even undesirable to give a circular definition. The same thing happens in defining the rules of English grammar, but there we seldom notice that it is circular. We can make two sentences out of one sentence by joining them with the word “and” or with a semicolon. Hence, if we were to write out the grammatical rules for English sentences, the definition of a sentence would refer to sentences and so would be recursive (that is, circular). For instance, a complete definition of English sentences would begin with something like “A *sentence* is any string of words formed according to the following rules.” One of the rules would read “A *sentence* can be formed by taking any *sentence* followed by the word ‘and’ followed by any other *sentence* (and, for written sentences, adjusting periods and capitalization appropriately).”

Recursive definitions require a bit of care in how they are formulated. In order for such a definition to be meaningful, it should contain some clauses that are not circular. In the case of Pascal statements, this general rule means that some statements may qualify for statementhood because some subpart (or subparts) is itself a statement, and that substatement may similarly contain a subpart that is a statement, and so forth. Eventually, however, this chain of substatement, sub-substatement, and so forth must bottom out with a statement that does not contain any substatements. This bottoming out is what saves these circular or recursive definitions from going on endlessly and meaninglessly when you try to use them. For example, an *if-then* statement can contain another *if-then* statement, and that *if-then* statement can contain yet another *if-then* statement, and so on for any number of times, but eventually this must end, and the innermost *if-then* statement must contain something simple like an assignment statement or a `writeln` statement.



*recursive  
(circular)  
definitions*

*how to  
formulate  
recursive  
definitions*

**Figure 6.1**  
Nesting an *if-then* statement within an *if-then* statement.

recursive  
syntax  
diagrams

The recursive nature of the definition of Pascal statements can be seen in their syntax diagrams. If you look at the syntax diagrams in Appendix 4, you will see that a statement can be formed in a number of ways. Some of these ways, such as `<if statement>`, refer to the diagram for `<statement>`. On the other hand, the diagrams for other clauses, such as `<assignment statement>` and `<write or writeln call>`, do not refer to the diagram labeled `<statement>` either directly or indirectly. When using syntax diagrams to verify the correctness of a statement, a successful check must ultimately involve one of these clauses that do not refer to the diagram labeled `<statement>`.

---

## Nesting If-Then and If-Then-Else Statements

When nesting *if-then* and *if-then-else* statements, the meaning of the nested statement may appear to be ambiguous. To illustrate the problem, consider the following scenario. The variable *P* contains an integer value representing some number of pennies. We want to design a statement that will output nothing if the value of *P* is zero and will output the value of *P*, correctly annotated with the word 'penny' or 'pennies', if the value of *P* is one or more than one. The following is one way to accomplish this goal:

```
if P > 0 then
    if P = 1 then
        writeln('one penny')
    else
        writeln(P, ' pennies')
```

By properly indenting the statements, we have masked the problem. To see the problem, write it as follows; the spacing is irrelevant to the compiler, and this rewriting should not affect the meaning.

```
if P > 0 then if P = 1
                then writeln('one penny')
                else writeln(P, 'pennies')
```

Written like this, it is not apparent which of the two *then*'s is supposed to be paired with the single *else*.

rules for  
pairing  
else  
with then

Pascal clarifies such ambiguities by specifying that an *else* is always paired with the closest preceding, unmatched *then*. So the meaning of our nested statement is what we intended.

---

## Complex Boolean Expressions

A Pascal expression that is either true or false is called a *boolean expression*. For example, we have been using simple boolean expressions such as `X > 0` within *if-then-else* statements. You can form more complicated boolean expressions out of these simple boolean expressions by combining them with the operators *and*, *or*,



and *not*. These operators work very much as they do in English, combining simpler boolean expressions to yield a new, complex boolean expression.

A larger boolean expression can be formed from two smaller boolean expressions by joining them with an *and*. The larger expression evaluates to *true* if both subexpressions evaluate to *true*; otherwise, it evaluates to *false*. For example,

```
(X < Y) and (Y < Z)
```

evaluates to *true* if both the value of *X* is less than the value of *Y* and the value of *Y* is less than the value of *Z*; otherwise, its value is *false*. In mathematics, pairs of inequalities such as the previous one are usually expressed as follows:

$$x < y < z$$

Expressions with such chains of interlocking comparisons are not allowed in Pascal. Instead, you must break them into parts and connect these parts with *and*'s.

These complex boolean expressions are used in the same way as the simple boolean expressions we introduced in Chapter 3. For example, the program in Figure 6.2 uses complex boolean expressions to determine if the user's zodiac sign is Scorpio.

The syntax for expressions involving *or* is similar to that described for *and*. For example

```
(X < Y) or (Y < Z)
```

This expression is *true* provided the value of *X* is less than the value of *Y* or the value of *Y* is less than the value of *Z* or *both*. This is the so-called "inclusive" meaning of "or." This inclusive meaning is always used in mathematics, but it is not always used in ordinary conversation. Ordinary conversational English has trouble coping with situations where two true statements are joined by the word "or." Pascal and mathematical disciplines in general have no such problem. In Pascal, if two things are connected by an *or*, then the resulting expression is *true* provided that one or both subexpressions are *true*; otherwise, its value is *false*.

The boolean operator *not* reverses truth values, much as the word "not" does in English. However, the Pascal syntax for *not* is quite different from English. In Pascal the *not* is always placed in front of the expression being negated. As examples, consider the following boolean expressions:

```
not ('A' = 'Z')
not (2 < 3)
```

Since *not* changes *true* to *false* and *false* to *true*, the first of these two boolean expressions evaluates to *true* and the second to *false*.

As the following two examples illustrate, we can repeat this method of combining boolean expressions with *and*, *or*, and *not* to obtain expressions that are even more complex.

```
((Month = 10) and (Day >= 24)) or ((Month = 11) and (Day <= 22))
(Time < 60) and not( (Ans = 'N') or (Ans = 'n') )
```

The first of these sample expressions could be used in a program similar to that in Figure 6.2. It would be *true* if the values of *Month* and *Day* indicate a Scorpio of any kind.

*and**or**not**nested  
expressions*



**Program**

```

program FindScorpios(input, output);
{Determines if user is a Scorpio: birthday between October 24th and November 22nd, inclusive.}
var Month, Day: integer;
begin{Program}
  writeln('Enter your month and day');
  writeln('of birth, as two numbers. ');
  readln(Month, Day);

  if (Month = 10) and (Day >= 24) then
    writeln('You're an October Scorpio. ')
  else
    writeln('You're not an October Scorpio. ');

  if (Month = 11) and (Day <= 22) then
    writeln('You're a November Scorpio. ')
  else
    writeln('You're not a November Scorpio. ');

  writeln('I knew it!')
end. {Program}

```

**Sample Dialogue 1**

```

Enter your month and day
of birth, as two numbers.
10 25
You're an October Scorpio.
You're not a November Scorpio.
I knew it!

```

**Sample Dialogue 2**

```

Enter your month and day
of birth, as two numbers.
10 21
You're not an October Scorpio.
You're not a November Scorpio.
I knew it!

```

**Figure 6.2**  
**Boolean**  
**expressions using**  
**and.**

*parentheses*

All the examples we have constructed thus far have been fully parenthesized to show exactly what two expressions each *and* or *or* applies to. This is not always required. The default precedence, if you omit parentheses, is as follows: *not* first, *and* second, and *or* third. However, it is good practice to include most parentheses in order to make the expression easier to understand. One place where parentheses can safely be omitted is a simple string of *and*'s or *or*'s, but not a mixture of the two. The following expression is acceptable both to the Pascal compiler and in terms of readability:

$(X < 100)$  and  $(Ans <> 'N')$  and  $(Ans <> 'n')$

When they are included in more complicated expressions, the parentheses around simple boolean expressions, such as  $(X < 100)$ , are never optional. If they are omitted, the compiler will either give an error statement or produce unwanted results.

Many high level programming languages have boolean expressions that are formed and used in much the same way as they are in Pascal. Minor details such as the placement of parentheses will vary from language to language, but the general ideas are the same for most programming languages.

---

## George Boole

### (Optional)

The word “boolean” is derived from the name of George Boole, a nineteenth-century English mathematician who developed the foundations for a formal calculus of such expressions. Boole was a self-educated scholar with limited formal training. He began his teaching career at the age of 16 as an elementary-school teacher and eventually progressed to a professorship at Queen’s College in Cork. He is considered by many to be the father of symbolic logic, and no less a commentator than Bertrand Russell has stated that “Pure mathematics was discovered by Boole, in a work which he called *The Laws of Thought*.”




---

## Evaluating Boolean Expressions

Boolean expressions such as

$(X > 0)$  and  $(Y < Z)$

are evaluated and have a value of either true or false. The computer obtains values for these boolean expressions in a way analogous to the way in which we (and the computer) normally evaluate arithmetic expressions.

By way of review, consider the following arithmetic expression:

$(1 + 2) * (2 + 3)$

To evaluate this expression, we evaluate the two sums to obtain the numbers 3 and 5. We then multiply the 3 and 5 to obtain 15 as the value of the entire expression. In doing the evaluation, we do not multiply the *expressions*  $(1 + 2)$  and  $(2 + 3)$ . Instead, we multiply the *values* of the expressions. We use 3. We do not use  $(1 + 2)$ .

The computer evaluates boolean expressions in a similar manner. Subexpressions are evaluated to obtain values each of which is either true or false. These values of true or false are then combined according to the rules in the table shown in Figure 6.3. For example, consider the boolean expression

$not ( (X > 0) or (X < 7) )$

and suppose that the value of X is 5. In this case,  $(X > 0)$  and  $(X < 7)$  both evaluate to true, and so the expression is equivalent to

$not (true or true)$

---

Expression	Value	Expression	Value
true <i>and</i> true	true	true <i>or</i> true	true
true <i>and</i> false	false	true <i>or</i> false	true
false <i>and</i> true	false	false <i>or</i> true	true
false <i>and</i> false	false	false <i>or</i> false	false
<i>not</i> (true)	false	<i>not</i> (false)	true

**Figure 6.3**  
Truth tables.

Consulting the table for *or*, the computer sees that the expression inside the parentheses evaluates to true and that the entire expression is therefore equivalent to *not*(true). Again consulting the tables, it sees that *not*(true) evaluates to false, and so it concludes that false is the value of the original boolean expression.

## Pitfall

### Undefined Boolean Expressions

Boolean expressions are evaluated by first evaluating the subexpressions and then combining those values, as we described in the previous section. This method of evaluation gives rise to one subtle problem. In many implementations of Pascal, if two subexpressions are connected by *and* or by *or*, the computer first evaluates *both* of these subexpressions and *then* uses these two values to determine the value of the full expression. This means that the two subexpressions must be well defined and capable of being evaluated. As an example, consider the following reasonable-looking statement:

```
if (Kids <> 0) and (Pieces div Kids >= 2) then
  writeln('Each child may have two pieces!')
```

If the value of Kids is not zero, this statement performs fine. However, suppose the value of Kids is zero. Then we might expect the boolean expression to evaluate to false. After all, the first subexpression evaluates to false, and using an *and* to combine false with any other value will yield a value of false. Unfortunately, the computer will try to evaluate *both* subexpressions *before* it applies the *and*. This will produce an error, since *div* is being asked to divide by zero, an error that can cause the program to terminate abnormally.

One way to avoid this problem is to use the following:

```
if Kids <> 0 then
  if Pieces div Kids >= 2 then
    writeln('Each child may have two pieces!')
```

In this version, the second boolean expression is not evaluated when the value of Kids is zero.

*undefined  
subexpressions*



The compilers for some other programming languages and even some Pascal compilers are smart enough to cope successfully with either of the above two statements. However, the first one should not be used, since it cannot be guaranteed to work.

## Self-Test Exercises

1. What output will be produced by the following code when embedded in a complete program?

```
writeln('Start');
if 2 <= 3 then
  if 0 <> 1 then
    writeln('First writeln')
  else
    writeln('Second writeln');
writeln('Next');
if 2 > 3 then
  if 0 = 1 then
    writeln('Third writeln')
  else
    writeln('Fourth writeln');
writeln('Enough')
```

2. Determine the value, true or false, of each of the following boolean expressions:

```
(0 = 1) and (2 < 3)           (0 = 1) or (2 < 3)
not( 0 = 1)                   'Y' = 'y'
('Y' = 'y') and (maxint = 65535)
not( (4.5 < 12.9) and (6 * 2 <= 13) )
not((31 mod 15) <> 1)
```

3. Translate the following English and mathematical expressions into Pascal boolean expressions: two plus two equals four; X plus seven is more than one hundred or else it is less than fifty; the value of the variable Z (of type char) is not one of the first three (uppercase) letters of the alphabet; X is not evenly divisible by 3; either X is not evenly divisible by 3 or Y is evenly divisible by 5;

$x \leq y + 2 \leq z$

---

He who would distinguish the true from  
the false must have an adequate idea of  
what is true and false.

*Benedict Spinoza, Ethics*

---

## Programming with Boolean Variables

*the type  
boolean*

The values `true` and `false` form a complete list of values for the data type called `boolean`. As we have seen, a boolean expression, like an arithmetic expression, yields a value. In the case of an arithmetic expression the value yielded is of type either `integer` or `real`. In the case of a boolean expression, the value is of type `boolean`. This value of type `boolean` can be stored in a variable, just as the value of an arithmetic expression can be stored in a variable, except that a variable that contains a value of `true` or `false` must be of type `boolean`. Hence, if `N` is a variable of type `integer` and `X` is a variable of type `boolean`, the following is perfectly meaningful in Pascal and sets the value of `X` equal to `false`:

```
N := 2;
X := (N > 10) or (N < 0)
```

*setting  
boolean  
variables*

At first, such boolean assignment statements look strange, but you quickly adjust to them. The rules are the same as they are for variables of other types: The expression on the right-hand side of the assignment operator is evaluated (since it is a boolean expression, its value will be either `true` or `false`); after that, the value of the boolean variable is set equal to this value of `true` or `false`. In the preceding example, the value of `N` is not greater than 10 nor is it less than 0. The boolean expression therefore evaluates to `false`, and the value of `X` is changed to `false`.

Variables of type `boolean` are declared in the same place and in the same way as other types of variables. A hypothetical program might start out as follows:

```
program Sample(input, output);
var Temperature: real;
    Ans: char;
    Raining, X: boolean;
```

The program might then contain the following statement:

```
Raining := (Ans = 'y') or (Ans = 'Y')
```

If the value of `Ans` is either `'Y'` or `'y'`, then this statement sets the value of `Raining` equal to `true`; otherwise, it sets it equal to `false`. Hence, this assignment statement is equivalent to the following longer and less efficient statement:

```
if (Ans = 'y') or (Ans = 'Y') then
    Raining := true
else
    Raining := false
```

The longer statement, which was given only to help explain the assignment statement, is poor programming style.

*why use  
boolean  
variables?*

Boolean variables can be used to remember a condition that may change later or that will not be easy to check later on. An equally important use of boolean variables is to make the meaning of a program more apparent. In the program fragment below, the boolean variable `Raining` frees the programmer from having to remember the exact

wording of the question 'Is it raining?' The question might have been 'Has it stopped raining?' and the programmer who forgets the exact wording of the question can easily misinterpret the meaning of `Ans`, even if the value of `Ans` does not change.

```
writeln('Is it raining?');  
readln(Ans);  
Raining := (Ans = 'y') or (Ans = 'Y');  
if Raining then  
    writeln('Too bad. ')  
else  
    writeln('Would you like to go for a walk?')
```

The use of boolean variables is illustrated in the program in Figure 6.4.

Many other common programming languages do not have boolean variables. In these languages, programmers frequently use some trick to simulate boolean variables, such as pretending that the integer value one means true and that zero means false.

---

## Pitfall

### Omitting Parentheses in Boolean Expressions

A boolean expression such as the following must include parentheses around simple comparisons like `X > Y`.

`(X > Y) and (Z > W)`

The reason for this has to do with the precedence rules for operations in the Pascal language. Unless parentheses indicate otherwise, the operations *and* and *or* are performed before comparisons such as `<` and `<>` are evaluated. Hence, the meaning of

`X > Y and Z > W`

is the puzzling-looking expression

`X > (Y and Z) > W`

which does not make sense. Although this boolean expression is incorrectly formed, the subexpression `(Y and Z)` is perfectly legal, provided both `Y` and `Z` were declared to be variables of type `boolean`. If you omit parentheses, the compiler will act as though `(Y and Z)` were a subexpression and will try to evaluate this subexpression. Moreover, it will do this even if the type of the variables is something other than `boolean`. Depending on details such as the type of the variables, a wide variety of things can then go wrong, but some error will surely be detected.



**Program**

```

program Talk(input, output);
var Ans: char;
    Raining: boolean;
begin{Program}
    writeln('Good day. My name is Ronald Gollum. ');
    writeln('I'm stuck in this box until quitting time. ');
    writeln('Please chat with me about the outside. ');

    writeln('Is it raining out now? ');
    readln(Ans);
    Raining := (Ans = 'Y') or (Ans = 'y');
    if not Raining then
        writeln('Too bad. We need rain. ');

    writeln('Do you think it will rain tomorrow? ');
    readln(Ans);
    if (Ans = 'Y') or (Ans = 'y') then
        writeln('I'll worry about that tomorrow. ');

    writeln('It's finally quitting time! Good bye. ');
    if Raining then
        writeln('You brightened up this rainy day. ')
    else
        writeln('I want to work on my tan. ')
end. {Program}

```

**Sample Dialogue 1**

Good day. My name is Ronald Gollum.  
 I'm stuck in this box until quitting time.  
 Please chat with me about the outside.  
 Is it raining out now?  
**no**  
 Too bad. We need rain.  
 Do you think it will rain tomorrow?  
**yes**  
 I'll worry about that tomorrow.  
 It's finally quitting time! Good bye.  
 I want to work on my tan.

**Sample Dialogue 2**

Good day. My name is Ronald Gollum.  
 I'm stuck in this box until quitting time.  
 Please chat with me about the outside.  
 Is it raining out now?  
**yes**

**Figure 6.4**  
**Program using a**  
**boolean variable.**

```
Do you think it will rain tomorrow?  
no  
It's finally quitting time! Good bye.  
You brightened up this rainy day.
```

**Figure 6.4**  
(continued)

---

## Boolean Input and Output

A value of type `boolean` cannot be read in directly. Instead, some other type of input, such as a character, must be read in and used to set the boolean variable. The following program fragment sets the value of the variable `BV` of type `boolean` using the variable `Ans` of type `char`:

```
writeln('Type t for True or f for False');  
readln(Ans);  
BV := (Ans = 't') or (Ans = 'T')
```

In many versions of Pascal, boolean values can be written as output; in others they cannot. In any version of Pascal, the following will serve to output the value of the boolean variable `BV`:

```
if BV then  
    write('true')  
else  
    write('false')
```

---

## Case Study

---

### Designing Output

#### Problem Definition

In Chapter 4 we created a program to determine the number of coins needed to give an amount of change from 1 to 99 cents. That program produced output such as the following:

```
    27 cents can be given as:  
1 quarters  
0 dimes and  
2 pennies
```

This sort of output is frequently acceptable, but it would be nicer to have output that is grammatically correct and that does not contain pointless information. Ideally, the output should look like the following:

---

```
27 cents can be given as:  
one quarter and 2 pennies
```

In this section we will design a procedure to produce this sort of output. We will design it for output that includes the possibility of nickels so that it would work for the enhanced version of the program that uses the redesigned procedures given in Figure 4.14. As in the original programming task, we will assume that the total amount of change is between 1 and 99 cents. The procedure will receive the amount and the coin counts as value parameters.

### Discussion

When you stop to think of all the grammatical details you take care of automatically when writing phrases such as the desired sample output, you quickly realize that designing output can be a complicated task. The program must somehow ignore coin amounts of zero, it needs to decide between singular and plural forms, it needs to place the 'and' correctly, and it needs to insert commas if three or more types of coins are used, as in

```
32 cents can be given as:  
one quarter, one nickel and 2 pennies
```

(There are two accepted ways of inserting commas in such expressions. To simplify the problem, we are using the rule that does not place a comma before the 'and'. )

#### subtasks

The value parameters for the total amount and for the counts of quarters, dimes, nickels, and pennies will be A, Q, D, N, and P. We will use a very straightforward decomposition of this task into subtasks:

*begin*

1. Write the heading.
2. If the number of quarters is not zero, then write it out.
3. If the number of dimes is not zero, then write it out preceded by a comma or 'and' if appropriate.
4. If the number of nickels is not zero, then write it out preceded by a comma or 'and', if appropriate.
5. If the number of pennies is not zero, then write it out preceded by a comma or 'and', if appropriate.

*end*

In order to determine whether it should output a comma or 'and', the program needs some way to test whether any coins were output previously. We will use a boolean variable `PreviousCoins`. This variable will be initialized to `false` and will have its value changed to `true` as soon as a nonzero number of coins is output. The value of `PreviousCoins` is part of the data that is passed from one subtask to another. A variable such as `PreviousCoins` that changes its value to indicate that some event has taken place is often called a *flag*. In this sample, when the flag “goes up,” it is time to insert a comma or 'and'. The pseudocode for outputting dimes is given below.

#### flags

---



*ALGORITHM*

```

{PreviousCoins is true if some number of quarters has been output.}
if D > 0 then
  begin; {Dimes > 0}
    if PreviousCoins then
      if there are more coins to follow then
        output a comma and appropriate spaces
      else {Dimes will be the last type of coin.}
        output the word 'and' and appropriate spaces;
    Write out the number of dimes;
    PreviousCoins := true
  end {Dimes > 0}

```

The test for more coins to follow can be accomplished with the boolean expression

$(N > 0) \text{ or } (P > 0)$

The complete procedure, embedded in a test program, is displayed in Figure 6.5. Since it is not used outside of the procedure, we made the boolean variable PreviousCoins a local variable.

---

## Boolean Constants and Debugging Switches

The two values of type boolean can be named using the two predefined constants *true* and *false*. These constants can be used anywhere that a boolean expression is allowed. So the following, although pointless, is allowed. It always causes the second `writeln` to be executed.

*true*  
*false*

```

if false then
  writeln('I know truth.')
else
  writeln('I know falsehood.')

```

Identifiers that have been given a boolean value in a constant declaration can be used in the same way, and their use is not always pointless. The following declaration makes `Debugging` a synonym for `true`.

```
const Debugging = true;
```

A boolean constant such as `Debugging` can be used as a switch to go between two forms of a program. To switch from one form of the program to the other, the named boolean constant in the constant declaration is changed from `true` to `false` or vice versa. This is particularly useful when debugging large programs.

Large programs often have trace statements and other debugging diagnostics written into the program. Often one wants to retain the debugging features even after the program is released for use, either because it will be changed or because you expect the users to discover new bugs. If the program is to be used, or even observed, without these debugging messages being sent to the screen, some method must be found for turning them off. One method is a boolean switch.

(continued, page 195)

---

**Program**

```

program Test(input, output);
var TotalAmount, Quarters, Dimes, Nickels, Pennies: integer;

procedure OutputCoins(A, Q, D, N, P: integer);
  {Outputs a collection of coins that total to A cents.
  Precondition:  $0 < A \leq 99$  and the coins total to A,
  that is,  $25*Q + 10*D + 5*N + P = A$ .}
  const Comma = ', '; {Includes space.}
  var PreviousCoins: boolean;
      {Set to true after the first output of a number of coins.}
  begin{OutputCoins}
    writeln(A:2, ' cents can be given as:');
    PreviousCoins := false;

    if Q > 0 then
      begin{Quarters > 0}
        if Q = 1 then
          write('one quarter')
        else
          write(Q:1, ' quarters');
        PreviousCoins := true
      end; {Quarters > 0}

    if D > 0 then
      begin{Dimes > 0}
        if PreviousCoins then
          if (N > 0) or (P > 0) then
            write(Comma)
          else {Dimes will be the last type of coin.}
            write(' and ');
        if D = 1 then
          write('one dime')
        else
          write(D:1, ' dimes');
        PreviousCoins := true
      end; {Dimes > 0}

    if N > 0 then
      begin{Nickels > 0}
        if PreviousCoins then
          if P > 0 then
            write(Comma)
          else {Nickels will be the last type of coin.}
            write(' and ');

```

**Figure 6.5**  
Enhanced  
procedure to  
output coins.

```

    if N = 1 then
        write('one nickel')
    else
        write(N:1, ' nickels');
    PreviousCoins := true
end; {Nickels > 0}

if P > 0 then
    begin{Pennies > 0}
        if PreviousCoins then
            write(' and ');
        if P = 1 then
            write('one penny')
        else
            write(Pennies:1, ' pennies')
        end; {Pennies > 0}
    writeln
end; {OutputCoins}

begin{TestProgram}
    writeln('Enter: Quarters, Dimes, Nickels, Pennies:');
    readln(Quarters, Dimes, Nickels, Pennies);
    TotalAmount := 25 * Quarters
                + 10 * Dimes
                + 5 * Nickels
                + Pennies;
    OutputCoins(TotalAmount, Quarters, Dimes, Nickels, Pennies)
end. {TestProgram}

```

### Sample Dialogue

```

Enter: Quarters, Dimes, Nickels, Pennies:
1 0 1 2
32 cents can be given as:
one quarter, one nickel and 2 pennies

```

**Figure 6.5**  
(continued)

If you include a constant declaration such as the sample one displayed for Debugging, then you can use it as a switch to turn on debugging statements like the following, which traces two variables:

```

if Debugging then
    begin{Debugging Trace}
        writeln('A = ', A, ' ALeft = ', ALeft);
        writeln('It should be true that ALeft = A mod 25')
    end {Debugging Trace}

```



To turn off the debugging features, simply reset the boolean constant by changing the constant declaration to

```
const Debugging = false;
```

### Program

```
program CaseSample(input, output);
var Grade: char;
begin{Program}
  writeln('What grade did you receive?');
  readln(Grade);
  case Grade of
    'A', 'B': writeln('Very good!');
    'C': writeln('Passing. ');
    'D', 'F': writeln('Too bad. ')
  end; {case}
  writeln('I have to go study. Goodbye. ')
end. {Program}
```

#### Sample Dialogue 1

```
What grade did you receive?
A
Very good!
I have to go study. Goodbye.
```

#### Sample Dialogue 2

```
What grade did you receive?
F
Too bad.
I have to go study. Goodbye.
```

**Figure 6.6**  
**Program with a**  
**case statement.**

## The Case Statement

An *if-then-else* statement chooses one of two alternative actions. A *case statement* is a different kind of Pascal statement that can choose from a list of any number of actions. A *case* statement is a complex statement made up of any number of simpler statements that serve as the alternative actions. When the *case* statement is executed, one (and only one) of the simpler statements is selected and executed. An example is likely to be more enlightening than an abstract discussion.

### example

Consider the program shown above in Figure 6.6. The five lines starting with the identifier *case* and ending with *end* contain a *case* statement (followed by a semicolon and a comment). The *case* statement contains three substatements: one labeled by the pair 'A', 'B'; one labeled by the constant 'C'; and one labeled by the pair 'D', 'F'. When the *case* statement is executed, exactly one of these three substatements will be executed; which one it is depends on the value of the variable *Grade*. When the *case* statement is executed, the value of *Grade* is checked and then the substatement labeled with that value is executed. Hence, if the value of *Grade* is either 'A' or 'B', the *writeln* following that pair will be executed. If

instead the value of `Grade` is 'C', the `writeln` following the 'C' will be executed. If the value of `Grade` is either 'D' or 'F', the third `writeln` is executed.

A *case* statement may have any number of alternatives from which to choose. Each alternative consists of exactly one statement. To obtain the effect of executing several statements for one alternative, you can use a compound statement, as shown in the program in Figure 6.7. Each alternative statement must be prefaced by a list of one or more constants, separated by commas and followed by a colon. The list of constants for a given statement is called a *label list*. The same constant may not appear in two label lists, since that would produce an ambiguous instruction. The alternative statements are separated by semicolons. You can add a semicolon after the final statement or not; the *case* statement meaning is the same either way. An *end* is used to terminate the *case* statement. There is no matching *begin* for this final *end*. Instead, it is matched to the reserved word *case*.

The expression that follows the word *case* need not be a variable. You can use any expression of type `integer` or `char` to control a *case* statement. (You can also use an expression of type `boolean`, but that is pointless, since an *if-then-else* statement could be used instead.) You can also use some other types, which are introduced later in this book. However, you cannot use an expression of type `real` to control a *case* statement. (This makes sense, since testing two `real` values for equality yields an unpredictable result.) When the *case* statement is executed, the expression is evaluated, and then the alternative labeled with that value is executed. Hence, the constants that label the alternative substatements must match this expression in type.

In standard Pascal you must ensure that the expression in a *case* statement evaluates to something that labels one of the alternatives. If it does not, then that is considered an error, and almost anything might happen. One way to defend against such an error is to embed the *case* statement in an *if-then* or *if-then-else* statement, as illustrated in Figure 6.7. As we will see in the next section, the TURBO Pascal *case* statement was designed to avoid this problem in a more graceful manner.

*syntax**label  
list**type of  
controlling  
expression*

## TURBO Pascal

### The Case Statement

In TURBO Pascal you need not ensure that the expression in a *case* statement evaluates to something that labels one of the alternatives. If it does not label one of the alternatives, then in TURBO Pascal nothing happens, and the program proceeds to the next statement. If the program in Figure 6.6 were run on a TURBO Pascal system, then the following would be a possible Sample Dialogue:

What grade did you receive?

P

I have to go study. Goodbye.

**Program**

```

program GiveDate(input, output);
{Tells registration dates. Works in both standard Pascal and TURBO Pascal.}
var Class: integer;
begin{Program}
    writeln('Enter your class code');
    writeln('and I will tell you when you can register. ');
    writeln('1 for freshman, 2 for sophomore, ');
    writeln('3 for junior, 4 for senior. ');
    writeln('Enter number (1, 2, 3, or 4): ');
    readln(Class);

    if (Class < 1) or (Class > 4) then
        writeln('Error: illegal class code! ')
    else
        case Class of
            1: begin{Freshman}
                writeln('Freshman: ');
                writeln('Enrollment for next year May 6-12')
            end; {Freshman}
            2,3: begin{Sophomores and Juniors}
                writeln('Sophomores and Juniors: ');
                writeln('Enrollment for next year May 1-5')
            end; {Sophomores and Juniors}
            4: writeln('Congratulations! ')
        end {case}
    end. {Program}

```

**Sample Dialogue 1**

```

Enter your class code
and I will tell you when you can register.
1 for freshman, 2 for sophomore,
3 for junior, 4 for senior.
Enter number (1, 2, 3, or 4):
0
Error: illegal class code!

```

**Sample Dialogue 2**

```

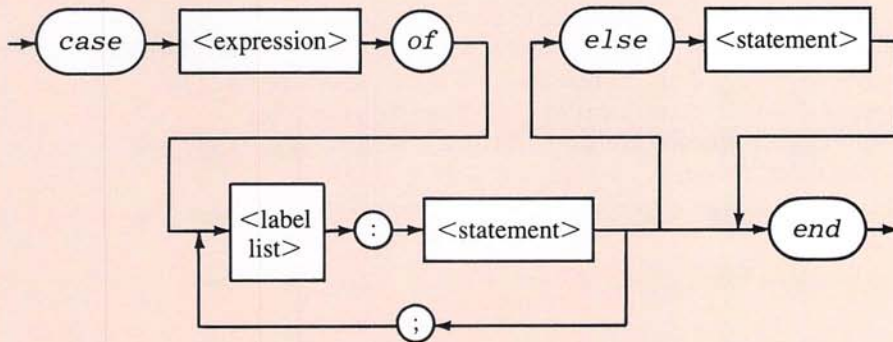
Enter your class code
and I will tell you when you can register.
1 for freshman, 2 for sophomore,
3 for junior, 4 for senior.
Enter number (1, 2, 3, or 4):
1
Freshman:
Enrollment for next year May 6-12

```

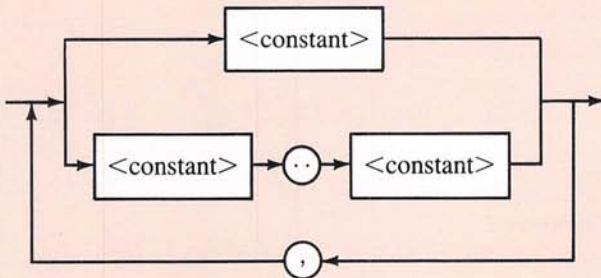
**Figure 6.7**  
Use of the **case**  
statement.



&lt;case statement&gt;



&lt;label list&gt;



**Figure 6.8**  
**Syntax of a TURBO**  
**Pascal case**  
**statement.**

In the above dialogue, the expression `Grade` evaluates to 'P', which labels none of the alternatives and so, in TURBO Pascal, nothing happens, and the program proceeds to the next statement, which happens to be a `writeln` statement.

As we have just seen, a TURBO Pascal *case* statement has an implicit instruction which says “do nothing if none of the label lists applies.” It is also possible to add explicit instructions stating something to be done when none of the label lists applies. When we write a TURBO Pascal *case* statement, it is possible to add a clause that says “Do the following if none of the other cases applies.” In TURBO Pascal this is done with an additional (optional) clause known as the *else clause*. If this clause is present and the value of the controlling expression is on no label list, then this *else* clause is executed. The syntax is given in Figure 6.8 and is illustrated by the following example:

(continued, page 201)

### Program

```

program GiveDate;
{Tells registration dates. Works in TURBO Pascal, but not in standard Pascal.}
var Class: integer;
begin{Program}
  writeln('Enter your class code');
  writeln('and I will tell you when you can register. ');
  writeln('1 for freshman, 2 for sophomore, ');
  writeln('3 for junior, 4 for senior. ');
  writeln('Enter number (1, 2, 3, or 4): ');
  readln(Class);

  case Class of
    1: begin{Freshman}
        writeln('Freshman: ');
        writeln('Enrollment for next year May 6-12')
      end; {Freshman}
    2,3: begin{Sophomores and Juniors}
        writeln('Sophomores and Juniors: ');
        writeln('Enrollment for next year May 1-5')
      end; {Sophomores and Juniors}
    4: writeln('Congratulations! ')
  else
    writeln('Error: illegal class code! ')
  end {case}
end. {Program}

```

### Sample Dialogue 1

```

Enter your class code
and I will tell you when you can register.
1 for freshman, 2 for sophomore,
3 for junior, 4 for senior.
Enter number (1, 2, 3, or 4):
0
Error: illegal class code!

```

### Sample Dialogue 2

```

Enter your class code
and I will tell you when you can register.
1 for freshman, 2 for sophomore,
3 for junior, 4 for senior.
Enter number (1, 2, 3, or 4):
1
Freshman:
Enrollment for next year May 6-12

```

Figure 6.7(t)  
A TURBO Pascal  
case statement.

```
case Grade of
  'A', 'B': writeln('Very Good!');
  'C': writeln('Passing. ');
  'D', 'F': writeln('Too bad. ')
else
  writeln('Not a possible grade. ')
end
```

A second example is given in Figure 6.7(t), which is a TURBO Pascal version of the program in Figure 6.7.

TURBO Pascal allows you to specify a range of values on a label list. For example, 2..5 is equivalent to the label list 2, 3, 4, 5. The range is expressed as the two end points separated by two periods. The following illustrates both the use of these ranges and the *else* clause:

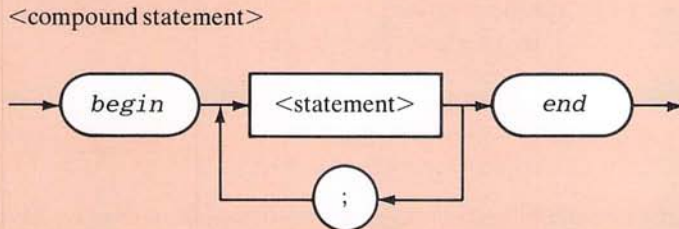
```
case Grade of
  90..100: writeln('Grade is A');
  80..89: writeln('Grade is B');
  70..79: writeln('Grade is C');
  60..69: writeln('Grade is D')
else
  writeln('Grade is F')
end
```

---

## The Empty Statement

Pascal has a statement that does absolutely nothing and is written by writing down absolutely nothing. It is called the *empty statement*, and our description makes it sound like a pure joke. Nonetheless, it does have a number of serious purposes. For one thing, it simplifies the syntax diagram for the compound statement.

Since a semicolon is used to separate statements, the last statement in a compound statement or in the body of a procedure or program should not have a semicolon after it. If you look at the syntax diagram for a compound statement in Figure 6.9, you will see that the diagram requires a statement, not a semicolon, just before the final *end*. However, we said earlier that including a final semicolon causes no harm. It seems that



**Figure 6.9**  
Syntax of the  
compound  
statement.



the syntax diagram needs to be complicated by adding another case for the possibility of that extra semicolon, but this complication is not needed. The empty statement is what allows for the extra semicolon. For example, the following compound statement satisfies the syntax diagram for a compound statement because a statement, namely the empty statement, is between the last semicolon and the *end*:

```
begin
  writeln('statement 1');
  writeln('statement 2');
end
```

The empty statement  
is here.

The empty statement is a formal trick that has the effect of sometimes allowing you to add semicolons. However, this trick only works in positions that allow you to insert a statement, such as just before the *end* in a compound statement. It does not allow you to insert semicolons in other places, such as before an *else*.

The empty statement can also be used in a *case* statement to specify that nothing be done in some alternative. In general, it can be used anywhere that you need a statement that causes no action and no changes.

---

## Programming Multiple Alternatives

The *case* statement chooses one of several statements to execute, but it is rather restricted in its use. The choice of which statement to execute must be made on the basis of a single value. Nested *if-then-else* statements are a more versatile way of implementing multiple alternative branches.

By way of example, suppose you are designing a game-playing program in which the user must guess the value of some number. The number can be named *Number* and the guess can be called *Guess*. If you wish to give a hint after each guess, you might design the following subalgorithm:

```
writeln('Too high. '), when Guess > Number
writeln('Too low. '), when Guess < Number
writeln('Correct! '), when Guess = Number
```

Any time a branching action is described as a list of mutually exclusive conditions and corresponding actions, as in this example, the branching action can be implemented in Pascal by using a nested *if-then-else* statement. For example, the above pseudocode translates to

```
if Guess > Number then
    writeln('Too high. ')
else if Guess < Number then
    writeln('Too low. ')
else if Guess = Number then
    writeln('Correct! ')
end if
```

The indenting pattern used here is slightly different from what we have advocated previously. If we had followed our indenting rules, we would have produced something like the following:

```

if Guess > Number then
    writeln('Too high. ')
else
    if Guess < Number then
        writeln('Too low. ')
    else {Guess = Number}
        writeln('Correct! ')

```

The first version, which violates our guidelines for indenting, is the one you should use. This is one of those rare cases in which our general guidelines for indenting nested statements should not be followed. The reason is that by lining up all the *else*'s, we also line up all the condition/action pairs and so make the layout of the program reflect our reasoning.

Since the conditions are mutually exclusive, the last *if* is superfluous and can be omitted, but it is usually best to include it in a comment as follows:

```

if Guess > Number then
    writeln('Too high. ')
else if Guess < Number then
    writeln('Too low. ')
else {Guess = Number}
    writeln('Correct! ')

```

---

## Case Study

---

### State Income Tax

#### Problem Definition

We will design a procedure to compute state income tax from net income (computed to the nearest dollar) according to the following formula:

1. No tax is paid on the first \$15,000 of net income.
2. A tax of 5% is assessed on each dollar of net income from \$15,001 to \$25,000.
3. A tax of 10% is assessed on each dollar of net income over \$25,000.

#### Discussion

If we let *FirstChunk* name the amount of income that is taxed at the 5% rate and let *SecondChunk* name the amount of income taxed at the 10% rate, then our algorithm to compute *FirstChunk* follows directly from the problem specification:

```

FirstChunk := 0, when NetIncome <= 15000
FirstChunk := NetIncome - 15000,
    when 15000 < NetIncome ≤ 25000
FirstChunk := 10000,
    when NetIncome > 25000;

```

ALGORITHM

---

```

procedure ComputeTax(NetIncome: integer; var Tax: real);
{Computes tax on NetIncome by rates: first $15000 no tax;
dollars from $15001 to $25000 at 5%; dollars over $25000 at 10%.}
var FirstChunk: integer; {Income to be taxed at 5%.}
    SecondChunk: integer; {Income to be taxed at 10%.}
begin{ComputeTax}
    if NetIncome <= 15000 then
        FirstChunk := 0
    else if (NetIncome > 15000) and (NetIncome <= 25000) then
        FirstChunk := NetIncome - 15000
    else {NetIncome > 25000}
        FirstChunk := 10000;

    if NetIncome <= 25000 then
        SecondChunk := 0
    else
        SecondChunk := NetIncome - 25000;

    Tax := (0.05 * FirstChunk) + (0.10 * SecondChunk)
end; {ComputeTax}

```

**Figure 6.10**  
**Procedure**  
**including**  
**multiple**  
**alternative**  
**actions.**

A similar piece of pseudocode can be written for computing SecondChunk. The total tax is then 5% of the first figure plus 10% of the second figure. Once the pseudocode has been written, it is routine to translate it into a Pascal procedure like the one in Figure 6.10.

---

## Case Study

---

### Scoring Blackjack

#### Problem Definition

In this section we will design a program to score a blackjack hand. Blackjack is a card game in which the players attempt to get a score that is as close as possible to a value of 21 without going over that value. Initially, a player is given two cards. In the usual version, a player can have as many additional cards as he or she wants. For the purposes of this example, we will assume that a player can have at most one additional card. Our program will assume that the cards have already been dealt and will compute the score for one player. Hence, the input will be either two or three card values. The output will be the player's score. Numeric cards are counted at face value; the jacks, queens, and kings count as 10; and aces are counted as either 1 or 11, whichever is better for the player. If the total exceeds 21, the player is said to be "busted." A busted player never wins. If two players have scores under 21, the highest score wins. Based on these scor-

---



ing rules, we see that the output should be either a number that is 21 or less or a message that the player is “busted.” Some examples may clarify the scoring rules. A hand with a 2, a 5, and a king should precipitate an output of 17. A hand of a king, a king, and a 5 would produce a message of “busted.” A hand of two 10’s and an ace can be scored as either 21 or 31, depending on whether the ace is counted as 1 or 11. Since a score of 31 means the player is “busted,” this hand would be scored as 21, and the program should output 21. A hand of an ace, a 5, and a 2 can be scored as either 8 or 18, depending on how the ace is counted. Since 18 is a better score for the player, the program should output 18.

## Discussion

We divide the task into three main subtasks, which will be implemented as three procedures:

1. **GetCards**: Have the user input two or three card values. Values 2 through 10 are entered as numbers. The other values are entered as words, such as “ace.”
2. **ChangeToNumber**: Convert the cards to a numeric value between 2 and 11. At this point, count aces as 11.
3. **OutputScore**: Output the score plus a message if the user is busted. Somehow adjust the values of any aces so as to benefit the player.

Since we are letting the user enter input as strings, such as 'King', we will read the input as some number of characters stored in variables of type `char`. Fortunately, each possible input is determined by its first character: 'Q' for 'Queen', '2' for '2', '1' for '10', and so forth. It is important to note that, for example, the number 2 is read in as the character '2' and not as a numeric value. This is because we have decided that the variables to receive the input are to be of type `char` some of the time, and hence they must be of that type all the time. Also notice that, since we only use one character, the number 10 is represented by '1'.

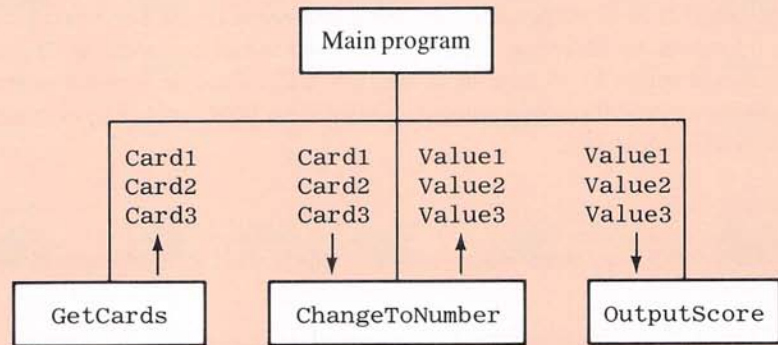
The parameters for the procedures can be determined from the data flow diagram in Figure 6.11. In that diagram we have assumed that there are three cards in the hand. If we can somehow interpret a hand of two cards as if it were three cards, we will not need to design a separate algorithm or separate parts of the algorithm for the case of two cards. One way to achieve this goal is to treat the two-card case as though it were three cards, one of which has a value of zero. Refining the subtask **GetCards** with this technique produces the following pseudocode:

```
begin{GetCards}
    Ask the user how many cards he or she has; (2 or 3)
    Ask for and read in Card1 ;
    Ask for and read in Card2 ;
    if the number of cards is 2 then
        Card3 := '0'
    else
        Ask for and read in Card3
end; {GetCards}
```

*data  
representation*

*unifying  
cases*

*ALGORITHM for  
Getcards*



#### Data Summary

Card1, Card2, Card3: three variables of type char to hold the card values. ('2' for 2, '3' for 3, . . . 'k' for king, 'a' for ace.)

Value1, Value2, Value3: three variables of type integer to hold the numeric values of the three cards. (Aces are stored as 11.)

**Figure 6.11**  
Data flow diagram  
for blackjack  
program.

Converting the cards to a numeric value is routine and is performed by the procedure `ChangeToNumber` shown in Figure 6.12. Since the same task is performed for each of the three cards, we have written one procedure that is called three times with different parameters each time. Notice that we have allowed the user to enter either upper- or lowercase letters by listing both upper- and lowercase letters in the label list. To save space, we have placed several of the case alternatives on a single line. The values are stored in three integer variables called `Value1`, `Value2`, and `Value3`; all aces are valued as 11 at this point. Later on the program will change some or all of the aces to 1 if that is to the advantage of the player.

Outputting the score is routine if the total (counting aces as 11) is 21 or less. This is done by the procedure `OutputScore`. The amount to output is more complicated if the total is over 21, since revaluing one or more aces to 1 could bring the total down to 21 or less and save the player from being “busted.” If the total value of the cards, counting aces as 11, is over 21, then a special procedure named `AcesAsOne` is called to handle the output. The pseudocode for that procedure follows. The total value of the hand, counting aces as 11, is stored in the variable `Total`.

```

{Precondition: Total > 21}
begin{AcesAsOne}
1. Count the number of aces (the number of values that are 11).
2. if there are no aces then
    writeln('Sorry, busted: you went over 21. ')
    else if there is exactly one ace then
        Output Total - 10 {which is always <= 21}
    else if there are exactly two aces then
        Output Total, possibly adjusted for one or two aces
    else if there are exactly three aces then
        writeln('Your score is 13')
end {AcesAsOne}

```

*ALGORITHM  
for AcesAsOne*

The score in the last case is easy to calculate since we know the exact hand, namely three aces. In the case of two aces, the procedure tries revaluing an ace as 1 instead of 11 to see if that helps the player. The effect of changing an 11 to a 1 is accomplished by subtracting 10 from Total. Once a player's score is 21 or less, all the remaining aces are left at the value of 11. The case of two aces is expanded into the following:

```

{Precondition: Total > 21 and there are exactly 2 aces.}
begin{AceCount = 2}
    if Total - 10 <= 21 then
        Output Total - 10
    else if (Total - 10 > 21) and (Total - 20 <= 21) then
        Output Total - 20
    else
        Output "busted"
    end {AceCount = 2}

```

Since we are assuming that there are at most three cards, the last case in the pseudocode can not occur. If there are two aces then the maximum score is 32 and that number can always be reduced to below 21 by subtracting one or two 10's. Hence, we can drop the last clause in the pseudocode. This simplifies the final Pascal code to a simple *if-then-else* as shown in the final version of *AcesAsOne*. That procedure and the entire program are displayed in Figure 6.12.

---

"Contrariwise," continued Tweedledee,  
 "if it was so, it might be;  
 and if it were so, it would be;  
 but as it isn't, it ain't. That's logic."

*Lewis Carroll, Through the Looking-Glass*

---



## Program

```

program Scorer(input, output);
{Scores a 2 or 3 card blackjack hand.}
var Card1, Card2, Card3: char;
    Value1, Value2, Value3: integer;

procedure GetCards(var Card1, Card2, Card3: char);
{Reads 2 or 3 cards as a CHARACTER code: '2'-'9' for 2-9; '1' for 10;
 'A' for Ace, etc.; if there are only 2 cards, then Card3 is set to '0'.}
var NumberOfCards: integer;
begin{GetCards}
    writeln('How many cards do you have, 2 or 3?');
    readln(NumberOfCards);
    writeln('Enter cards as either 2 through 10,');
    writeln('Jack, Queen, King, or Ace. ');
    writeln('Enter your first card: ');
    readln(Card1);
    writeln('Enter your second card: ');
    readln(Card2);
    if NumberOfCards = 2 then
        Card3 := '0'
    else
        begin{input third card}
            writeln('Enter your third card: ');
            readln(Card3)
        end {input third card}
    end; {GetCards}

procedure ChangeToNumber(Card: char; var Value: integer);
{Precondition: The value of Card is one of: '0' through '9', 'J', 'Q', 'K',
 'A' (or their lowercase version). Postcondition: Value is equal to the numeric
 equivalent of Card; '1' is valued at 10; 'A' at 11; 'J', 'Q', 'K' at 10.}
begin{ChangeToNumber}
    case Card of
        '2': Value := 2; '3': Value := 3; '4': Value := 4;
        '5': Value := 5; '6': Value := 6; '7': Value := 7;
        '8': Value := 8; '9': Value := 9;
        '1', 'J', 'j', 'Q', 'q', 'K', 'k': Value := 10;
        'A', 'a': Value := 11;
        '0': Value := 0
    end{case}
end; {ChangeToNumber}

```

**Figure 6.12**  
Program using  
branching.

```

procedure AcesAsOne(Value1, Value2,
                    Value3, Total: integer);
{Precondition: Total = Value1 + Value2 + Value3 and Total > 21. All the
conditions for the procedure OutputScore also apply to this procedure.}
var AceCount: integer;
begin{AcesAsOne}
  AceCount := 0;
  if Value1 = 11 then
    AceCount := AceCount + 1;
  if Value2 = 11 then
    AceCount := AceCount + 1;
  if Value3 = 11 then
    AceCount := AceCount + 1;

  {AceCount is the number of aces in the hand; Total is
the total score, counting aces as 11; Total > 21.}

  if AceCount = 0 then
    writeln('Sorry, busted: you went over 21.')
  else if AceCount = 1 then
    writeln('Your score is ', (Total - 10):2)
  else if AceCount = 2 then
    if Total - 10 <= 21 then
      writeln('Your score is ', (Total - 10):2)
    else
      writeln('Your score is ', (Total - 20):2)
  else {AceCount = 3}
    writeln('Your score is 13')
end; {AcesAsOne}

procedure OutputScore(Value1, Value2, Value3: integer);
{Precondition: Parameters are the numeric values of cards; Ace has value
11, face cards have value 10. Postcondition: The score is output to the
screen; Aces are counted as 1 if that is needed to bring the score below 21.}
var Total: integer;
begin{OutputScore}
  Total := Value1 + Value2 + Value3;
  {Total contains the hand score, counting all aces as 11.}
  if Total <= 21 then
    writeln('Your score is ', Total:2)
  else
    AcesAsOne(Value1, Value2, Value3, Total)
end; {OutputScore}

```

**Figure 6.12**  
(continued)

```
begin{Program}  
  writeln('This program scores a two or three card');  
  writeln('blackjack hand. ');  
  writeln('Jack, Queen, King count as 10. ');  
  writeln('Ace counts as either 1 or 11. ');  
  writeln('If you exceed 21 you are "busted." ');  
  GetCards(Card1, Card2, Card3);  
  ChangeToNumber(Card1, Value1);  
  ChangeToNumber(Card2, Value2);  
  ChangeToNumber(Card3, Value3);  
  OutputScore(Value1, Value2, Value3)  
end. {Program}
```

### Sample Dialogue

```
This program scores a two or three card  
blackjack hand.  
Jack, Queen, King count as 10.  
Ace counts as either 1 or 11.  
If you exceed 21 you are "busted."  
How many cards do you have, 2 or 3?  
3  
Enter cards as either 2 through 10,  
Jack, Queen, King, or Ace.  
Enter your first card:  
Ace  
Enter your second card:  
5  
Enter your third card:  
ace  
Your score is 17
```

**Figure 6.12**  
(continued)

---

## Summary of Problem Solving and Programming Techniques

- A boolean variable can be used within a program as a flag to record whether or not some specific action has taken place.
  - A boolean constant can be used as a switch to turn program features off and on by changing the constant declaration. One such feature is trace statements for debugging.
  - One approach to solving a task or subtask is to write down conditions and the corresponding actions that need to be taken under each condition. This can be implemented in Pascal as a series of nested *if-then-else* statements.
-



- When designing programs, consider the possibility of alternative data types. For example, in the blackjack program we read numeric card values as if they were characters and thereby simplified the input routine.
- When designing an algorithm, if you can unify two or more cases into one more general case, then you can sometimes simplify the algorithm. For example, when we scored a numeric card hand that could contain either two or three cards, we treated the two-card hand as a three-card hand with one zero-valued card.

---

## Summary of Pascal Constructs

*if-then* statements, *if-then-else* statements, and simple boolean expressions are summarized at the end of Chapter 3.

### the type **boolean**

Syntax:

```
boolean
```

Example:

```
var Raining: boolean;
```

A Pascal type. There are exactly two values of this type, namely **true** and **false**. A Pascal program can have variables, constants, and expressions of type **boolean**.

### use of **and**

Syntax:

```
<boolean expression 1> and <boolean expression 2>
```

Examples:

```
(X < 1) and (X <> Y)
(sqrt(X) < 10.7) and (Y > 0)
```

One way to make a larger boolean expression out of two smaller boolean expressions. If both *<boolean expression 1>* and *<boolean expression 2>* evaluate to **true**, then the entire expression evaluates to **true**. If at least one of *<boolean expression 1>* and *<boolean expression 2>* evaluates to **false**, then the entire expression evaluates to **false**. (It is sometimes necessary, and it is usually wise, to place parentheses around the subexpressions.)

### use of **or**

Syntax:

```
<boolean expression 1> or <boolean expression 2>
```

Example:

```
(Ans = 'Y') or (Number = 7)
```

---

One way to make a larger boolean expression out of two smaller boolean expressions. If at least one of <boolean expression 1> and <boolean expression 2> evaluate to true, then the entire expression evaluates to true. If both <boolean expression 1> and <boolean expression 2> evaluate to false, then the entire expression evaluates to false. (It is sometimes necessary, and it is usually wise, to place parentheses around the subexpressions.)

### use of not

Syntax:

```
not (<boolean expression>)
```

Examples:

```
not (X < 0)
not ( (X < 1) and (Y < 0) )
```

One way to make a larger boolean expression out of a smaller one. *not* reverses boolean values. If <boolean expression> evaluates to true, then the expression with the *not* evaluates to false. If <boolean expression> evaluates to false, then the expression with the *not* evaluates to true. If <boolean expression> is a boolean variable (or certain other boolean expressions described later in this book), parentheses are not required around the expression.

### case statement

Syntax:

```
case <expression> of
  <label list 1> : <statement 1>;
  <label list 2> : <statement 2>;
  .
  .
  .
  <label list n> : <statement n>
end
```

Example:

```
case N of
  2: writeln('Value of N is 2');
  7,9,4: writeln('Value of N is 7, 9, or 4')
end
```

<expression> must evaluate to a value of type integer or type char. (Later we will discover other allowable types.) It cannot be of type real or string. The label lists must be lists of values of the same type as that of <expression>, and all these values must be different. The statements may be any Pascal statements. When the *case* statement is executed, <expression> is evaluated and the statement with that value on its label list is executed. TURBO Pascal allows an optional *else* clause.

---

## Exercises

### Self-Test Exercises

4. The expression `A or B` evaluates to `true` provided that the value of either *or both* of the variables is `true`. Design a boolean expression that evaluates to `true` provided that the value of *exactly one* of the two variables is `true`.
5. What is the output produced by the following code when embedded in a complete program in which `X` is declared to be a variable of type `boolean`?

```
if true then
    writeln('First writeln')
else
    writeln('Second writeln');
X := (1 < 2) and (4 < 3);
if X then
    writeln('Third writeln')
else
    writeln('Fourth writeln')
```

6. Write a program that reads in three integers and outputs a message telling whether or not they are in numeric order.
7. Write a nested *if-then-else* statement that classifies an integer `X` into one of the following categories and writes out an appropriate message:

`X < 0` or `0 ≤ X ≤ 100` or `X > 100`

8. We have seen five types thus far in our discussion of the Pascal language: `integer`, `real`, `char`, *string*, and `boolean`. Which of these types are allowed as the type for the controlling expression in a *case* statement? That is, what can the type of `<expression>` be in a *case* statement that begins with

`case <expression> of`

9. Write a program the input to which is a month entered as a number from 1 to 12 and the output of which is the number of days in that month.
10. The following four boolean expressions divide into two groups with two equivalent expressions in each group. What are the two groups? (Two expressions are equivalent if they evaluate to the same value of `true` or `false` for each possible way of setting `FootLoose` and `FancyFree` to `true` or `false`.)

```
not(FootLoose) and not(FancyFree)
not(FootLoose) or not(FancyFree)
not(FootLoose and FancyFree)
not(FootLoose or FancyFree)
```



### Interactive Exercises

11. Embed the following code in a complete program and run it several times using different input values each time:

```
writeln('Type in three integers: ');
readln(X, Y, Z);

if X > 0 then
  writeln('X is greater than zero')
else if Y > 0 then
  writeln('Y is greater than zero')
else if Z > 0 then
  writeln('Z is greater than zero')
else
  writeln('They are all zero or negative')
```

Predict the outputs before you run the program.

12. Write a program that will read in three integers and output a message telling whether exactly two of them are greater than 10. There is no need to be fancy. It is perfectly all right to use a long boolean expression that tests all possible pairs of variables.

13. Write a program whose input is a one-digit number and whose output is that number written as a word. For example, an input of 5 should produce an output of: *five*.

### Programming Exercises

14. Enhance the program from Exercise 9 so that the month is input as a name, such as "January," and not as a number. If the month is February, the program also asks the year in order to determine whether it is a leap year. The program in Figure 5.20 can be modified into a procedure to determine whether a year is a leap year. The program should allow the month to be spelled with any combination of upper- and lowercase letters. It should accept misspelled months as long as the first three letters are correct.

15. Write a program that computes state income tax according to the following formula: Net income is gross income minus deductions (both given as input); tax is:

```
3% on each dollar of net income up to $8,000 plus
5% on each dollar of net income from $8,001 to 15,000 plus
8% on each dollar of net income over $15,000
```

16. Write a program that guesses the user's height. The program makes a first guess and then asks the user if it is too high, too low, or correct. The program continues to guess and ask the user for feedback until the user says the guess is correct or until three tries have been made, whichever comes first.

17. Write a program to determine grades in a course with three quizzes, each scored on a basis of ten points. Grades are determined according to the following rule: A is an average of 9.0 or better; B is an average of 8.0 or better (up to 9.0); C is an average of 7.0 or better; D is an average of 6.0 or better; and less than 6.0 is an F.

---

18. A bicycle salesperson is offered a choice of wage plans: (1) a straight salary of \$300 per week; (2) \$3.50 per hour for 40 hours plus a 10% commission on sales; (3) a straight 15% commission on sales with no other salary. Write a program that takes as input the salesperson's expected weekly sales and outputs the wages paid under each plan as well as announcing the best-paying plan.

19. Write a program to compute the interest due, total amount due, and the minimum payment for a revolving credit account. The program accepts the account balance as input and then adds on the interest to get the total amount due. The rate schedules are as follows: The interest is 1.5% on the first \$1,000 and 1% on any amount over that. The minimum payment is the total amount due if that amount is \$10 or less; otherwise, it is \$10 or 10% of the total amount owed, whichever is larger.

20. Write a program to give a student's final grade in a course with the following grading scheme: there are three quizzes worth 10 points each, a midterm worth 100 points, and a final worth 100 points. The grade is based on the following weights: 50% on the final exam, 25% on the midterm, and 25% on the quiz average. (Be sure to normalize the quiz average to 100 by multiplying by 10.) The grade is determined from the weighted average in the traditional way: 90 or over is an A, below 90 down to 80 is a B, below 80 to 70 is a C, below 70 to 60 is a D, and below 60 is an F.

21. Write a program that computes the cost of postage on a first-class letter according to the following rate scale: 24 cents for the first ounce or fraction of an ounce and 7 cents for each additional half-ounce, plus a \$5 service charge if the customer desires special delivery.

22. Write a program that reads in a time of day in 24-hour notation and outputs it in 12-hour notation. For example, if the input is 13:45, the output should be

1: 45 PM

The program should instruct the user always to enter exactly five characters. So, for example, nine o'clock should be input as

09:00

23. Write a program that accepts dates written in the usual way and then outputs them as three numbers. For example, the input

February 15, 1961

should produce the output

2 15 61

24. Write a program that accepts a "three-digit" number written in words and then outputs it as a value of type `integer`. The input is terminated with a period. For example, the input

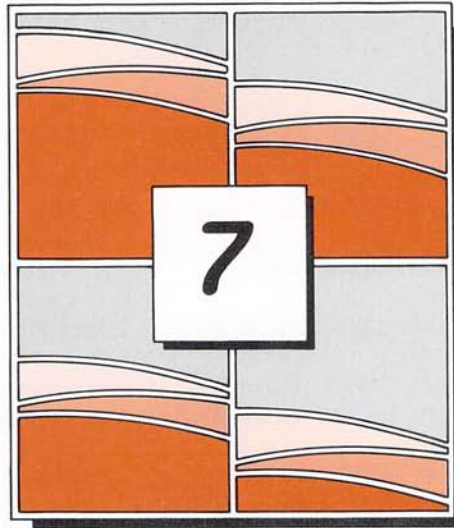
two hundred thirty-five.

should produce the output

235

25. Write a program that reads in the radius of a circle and then outputs one of the following depending on what the user requests: circumference of the circle, area of the circle, or diameter of the circle.
  26. Write a program to score the “paper-rock-scissors” game. Each of two users types in 'P', 'R' or 'S', and the program announces the winner as well as the basis for determining the winner: “paper covers rock,” “rock breaks scissors,” “scissors cut paper,” or “nobody wins.” Be sure to allow the users to use lower- as well as uppercase letters (if they are available on your machine).
  27. Write a program that accepts a year written as a four-digit Arabic (ordinary) numeral and outputs the year written in Roman numerals. Important Roman numerals are V for 5, X for 10, L for 50, C for 100, D for 500, and M for 1000. Recall that some numbers are formed by using a kind of subtraction of one Roman “digit”; for example, IV is 4, produced as V minus I; XL is 40; CM is 900, and so forth. Here are a few more sample years: MCM is 1900, MCML is 1950, MCMLX is 1960, MCMXL is 1940, and MCMLXXXIX is 1989. Assume the year is between 1000 and 3000.
  28. Write an astrology program. The user types in his or her birthday, and the program responds with the user's sign and horoscope. The month may be entered as a number from 1 to 12. Use a newspaper horoscope section for the horoscopes and dates of each sign. Then enhance your program so that if the user is only one or two days away from an adjacent sign, the program announces that the user is on a “cusp” and also outputs the horoscope for the nearest adjacent sign. For a nicer but more difficult program, let the user type in the month as a word rather than as a number.
  29. Write a program that accepts a number written as a Roman numeral and outputs the equivalent Arabic (ordinary) numeral. Assume that the Roman numeral is 50 or smaller (Roman numeral L or smaller.) See Exercise 27 for a review of Roman numerals.
  30. Enhance the blackjack program in Figure 6.12 to add the following features: The user may enter up to five cards per hand; the program scores two hands and announces the winner as well as the two scores; if both players have 21 and one player used five cards while the other used four cards or less, then the hand with five cards wins. (“Five cards under 21” is a hard hand to get and hence given more value.)
-





## *Problem Solving Using Loops*

It is not true that life is one  
damn thing after another—  
It's one damn thing over and over.

*Edna St. Vincent Millay*

## Chapter Contents

A Case Study Introducing Loops	Case Study—Finding the Largest and Smallest Values on a List
The While Statement	Unrolling a Loop
Pitfall—Uninitialized Variables	Designing Robust Programs
Pitfall—Unintended Infinite Loops	Invariant Assertions and Variant Expressions (Optional)
Self-Test Exercises	For Statements
Terminating an Input Loop	Example—Summing a Series
EOLN	Repeat N Times
Pitfall—Use of EOLN with Numeric Data	Invariant Assertions and For Loops (Optional)
Reading from a File (Optional)	What Kind of Loop to Use
Off-Line Data and a Preview of EOF (Optional)	Nested Loops
Modifying an Algorithm	Debugging Loops
The Repeat Statement	Case Study—Calendar Display
Comparison of the While and Repeat Loops	Starting Over
TURBO Pascal—Str and Val (Optional)	Summary of Problem Solving and Programming Techniques
Self-Test Exercises	Summary of Pascal Constructs
Case Study—Testing a Procedure	Exercises
	References for Further Reading

A very common sort of instruction in an algorithm is to repeat some action a number of times. A program that reads in a list of numbers may repeat the same sequence of prompt and read statements until all the numbers are read in. A program to calculate change may repeat its entire calculation, computing change for different amounts, until the user is through with the program. A program to update an inventory list may repeat the update process as many times as the user specifies.

Any program instruction that repeats some statement or sequence of statements a number of times is called a *loop*. In this chapter we will introduce the looping mechanisms available in the Pascal language. We will also describe a number of techniques for designing algorithms and programs that use loops.

---

## A Case Study Introducing Loops

Suppose we wish to sum a list of numbers typed in at the keyboard—homework scores or sales figures, for example. The obvious way to accomplish this is to read in the numbers and keep a running total of all the numbers seen so far. To hold this running sum, we use a variable called *Sum*. The variable *Sum* is set equal to zero, and then each time the program reads in a number, *Sum* is increased by adding in that number. After initializing *Sum* to zero, the program will perform some action equivalent to the following:

```
read (Number) ;
Sum := Sum + Number;
read (Number) ;
Sum := Sum + Number;
read (Number) ;
Sum := Sum + Number;
and so forth for each number in the list.
```

In other words, we want the program to repeat the following for each number to be added in:

```
read (Number) ;
Sum := Sum + Number;
```

Such repeated actions are called *loops*. We will begin introducing the Pascal constructs for realizing loops in the next section, but first we will discuss the concept of looping independent of any particular Pascal syntax.

The action (or actions) to be repeated in a loop is called the *body* of the loop, and each repetition of the loop body is called an *iteration*. The two main design questions to ask when constructing loops are, What should the body be? and How many times should it be iterated? To illustrate these two concepts, let us return to our example of summing a list of numbers.

When using a loop to add a list of numbers, we need to somehow determine how many numbers we need to add. We need some way to “stop” the loop. In this first example, we will use a very simple strategy: We will first ask the user how many numbers there are and then iterate the loop that number of times. In order to count the number of iterations, we will use a variable called *LeftToRead*, which will be initialized to the number of numbers to be read in and will then be decreased by one each time the loop is iterated. This adds one more statement to the loop body:

```
LeftToRead := LeftToRead - 1
```

The complete pseudocode for our algorithm, which includes this loop, is given in Figure 7.1. The next section describes how this loop can be realized within a Pascal program.

*loop body*  
*loop iteration*



### Algorithm to Add N Numbers

```
writeln('How many integers will there be in the list?');
Read number into variable N;
LeftToRead := N;
Sum := 0;
{The loop part of the algorithm starts here.}
Do the following, provided LeftToRead > 0, and continue to
do it again and again as long as LeftToRead > 0:
  begin{loop}
    read(Number);
    Sum := Sum + Number;
    LeftToRead := LeftToRead - 1
  end; {loop}
{The loop part of the algorithm ends here.}
writeln('The sum of the ', N, ' numbers is ', Sum)
```

**Figure 7.1**  
Pseudocode for a  
loop.

## The While Statement

The loop in Figure 7.1 is not the correct syntax for any Pascal construct. However, there is a Pascal statement that can exactly express the action specified by that pseudocode loop. The statement is called a *while* statement, and the correct syntax for the *while* statement corresponding to our pseudocode is as follows:

```
while LeftToRead > 0 do
  begin
    read(Number);
    Sum := Sum + Number;
    LeftToRead := LeftToRead - 1
  end
```

A complete program that is equivalent to the algorithm in Figure 7.1 is given in Figure 7.2.

*syntax*  
*body of a while loop*

The syntax of the *while* statement is shown in Figure 7.3. It consists of the word *while* followed by a boolean expression, followed by the word *do*, followed by a Pascal statement. This statement is the body of the *while* loop. As in our example, the statement that forms the body may be a compound statement. When the *while* loop is executed, this statement will be executed some number of times, depending on the values of the boolean expression.

*action of a while statement*

When the *while* statement is executed, the first thing that happens is that the boolean expression is evaluated. If it evaluates to *false*, then no action is taken and the program proceeds to the next statement after the loop. If the boolean expression evaluates to *true*, then the body of the loop is executed, after which the boolean expression is again evaluated. This process is iterated again and again as long as the boolean expression remains *true*. After each iteration the boolean expression is

**Program**

```

program AddUp(input, output);
{Computes the sum of a list of integers entered as input.}
var N, LeftToRead, Number, Sum: integer;
begin{Program}
  writeln('This program adds a list of integers. ');
  writeln('How many integers will there be in the list? ');
  readln(N);
  writeln('Now type in the ', N, ' integers. ');

  LeftToRead := N;
  Sum := 0;
  while LeftToRead > 0 do
    begin{while}
      read(Number);
      Sum := Sum + Number;
      LeftToRead := LeftToRead - 1
    end; {while}

  writeln('The sum of the ', N, ' numbers is ', Sum)
end. {Program}

```

**Sample Dialogue 1**

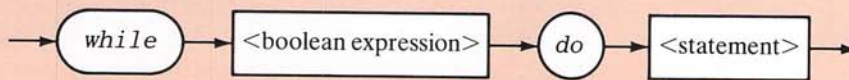
This program adds a list of integers.  
 How many integers will there be in the list?  
 3  
 Now type in the 3 integers.  
 4 6 2  
 The sum of the 3 numbers is 12

**Sample Dialogue 2**

This program adds a list of integers.  
 How many integers will there be in the list?  
 0  
 Now type in the 0 integers.  
 The sum of the 0 numbers is 0

**Figure 7.2**  
 Program with a  
*while* loop.

<while statement>



**Figure 7.3**  
 Syntax of the  
*while* statement.

again checked and, if it is true, the loop is iterated again; if it has changed from true to false, the *while* statement ends and the program proceeds to the next statement in the program.

As a concrete example, consider the *while* loop in Figure 7.2. The first thing that happens when that statement is executed is that the boolean expression

```
LeftToRead > 0
```

is evaluated. If it is false—in other words, if the value of `LeftToRead` is zero or less—then the *while* loop terminates without performing any action whatsoever, and the program proceeds to the `writeln` statement. On the other hand, if the value of `LeftToRead` is greater than zero, then the loop body is executed. This changes the value of `LeftToRead`. After the loop body is executed, the boolean expression is again evaluated. If the value of `LeftToRead` is now zero or less, the process ends; if the value of `LeftToRead` is still greater than zero, the loop body is executed again. This process continues as long as the boolean expression evaluates to true. When the boolean expression evaluates to false (that is, when `LeftToRead` is zero or less), the *while* loop terminates, and the program proceeds to the following `writeln` statement.

executing  
the body  
zero times

Notice that the first thing that happens when a *while* loop is executed is that the boolean expression is evaluated; if it evaluates to false at that point, then the body of the loop is never executed at all. It may seem pointless to execute the body of a loop zero times. After all, it will have no effect at all on any values or output. However, it is sometimes the desired action. If our example program were adding a list of homework scores, it might turn out that some student did absolutely no homework, and so the list of scores would be empty. To get the correct sum (and grade) of zero, the loop must be executed zero times.

Also notice that in our example, the body of the *while* loop changed, or at least had the potential for changing, the value of the boolean expression. This is the only way that the *while* loop can be ended. If the boolean expression has the value true and the loop body never changes the value of the boolean expression, then the loop will never end.

---

## Pitfall

### Uninitialized Variables

The program in Figure 7.2 sums a list of numbers by adding one number to the value of `Sum` during each loop iteration. Changing one or more variables in some incremental way is a typical action performed in a loop body. When designing such a loop, it is easy to become so concerned with the design of the loop body that you forget to initialize the variables used in the loop. A statement such as the following will produce unpredictable results unless the variable `Sum` is first given a value:

```
Sum := Sum + Number
```

---



If this statement is used in a loop to sum a list of numbers, then the variable `Sum` should be initialized to zero before the loop is executed. To see that zero is the correct initial value, simply note that zero plus the first number produces the correct value for a sum of one number.

Do not assume that zero is always the correct initial value for a variable used in a loop. Different situations require different initial values. For instance, the following computes the product of a list of `N` numbers:

```
LeftToRead := N;
Product := 1;
while LeftToRead > 0 do
  begin{while}
    read (Number);
    Product := Product * Number;
    LeftToRead := LeftToRead - 1
  end {while}
```

To see that `Product` should be initialized to one, note that one multiplied by the first number yields the correct starting value. Any other value would produce problems. In particular, if `Product` were initialized to zero, its value would remain zero no matter how many times the loop were iterated.

What happens when you fail to initialize a variable varies from system to system, but it almost always produces problems. Even worse, the problem may not be apparent. One common situation is that uninitialized variables simply receive some leftover value stored in memory by a previously run program. This sets the value of the variables to some unpredictable values. The values may even vary from one run of the program to another. Any time that a program is run twice with *absolutely* no changes and with *identical* input and yet produces different outputs, suspect an uninitialized variable of some sort. The problem can even occur if there is no loop. Every variable must be explicitly given a value before it can appear on the right-hand side of an assignment statement or anywhere else that expects it to already have a value.

## Pitfall

### Unintended Infinite Loops

Some loops are not required to end. For example, an airline reservation system might simply repeat a loop that allows the user to add or delete reservations. The program and the loop run forever, or at least until the computer breaks or the airline goes bankrupt. More often, a loop is designed to compute a value or a small list of values and should end after finding the value(s). For example, our sample loop to add up a list of numbers ends after summing all the numbers. A loop that repeats forever is called an *infinite loop*. An unintended infinite loop is a common error that should be guarded against.

Consider the following loop, which displays the interest produced by the value of `Amount` for some sample interest rates in the range from 10% to 20%:

```
Rate := 10;
while Rate <> 20 do
begin{while}
  Interest := Rate * 0.01 * Amount;
  {The 0.01 changes percent to a fraction.}
  writeln(Rate, '% yields $', Interest,
          ' in interest. ');
  Rate := Rate + 2
end {while}
```

Now suppose that we wish to change the display so that it shows changes of 3% rather than 2%. If we change the last line of the loop body to the following, we will produce an infinite loop:

```
Rate := Rate + 3
```

The problem is that the value of `Rate` now jumps from 19 to 22 and is never equal to 20. So the value of the boolean expression is never changed to `false`. The correct boolean expression is

```
Rate < 20
```

It would be safer to use this expression even in the original version of the loop, which increased percentages by 2 and thus happened to terminate correctly. With the change, the loop is robust enough to perform correctly even if we need to alter it slightly.

As a general rule, it is safer to terminate a loop with a test that involves a greater-than or less-than comparison rather than a test for exact equality or a test using the inequality operator `<>`. In the case of `real` values, this is an absolute rule. Since `real` values are stored as approximate quantities, a test for equality of `real` values is meaningless. Controlling a loop with a boolean expression that tests two `real` values for equality is virtually guaranteed to end the loop too soon or not at all. Always arrange to test `real` values using one of the relations `<`, `<=`, `>`, or `>=`.

*dangers  
of equality*

## Self-Test Exercises

1. What is the output of the following (when embedded in a correct program with `X` declared to be of type `integer`)?

```
X := 10;
while X > 0 do
  X := X - 3;
writeln(X)
```

2. What output would be produced by the code in Exercise 1 if the boolean expression were changed to  $X < 0$ ?
3. What is the output of the following (when embedded in a correct program with  $X$  declared to be of type integer?)

```
X := 10;
while X > 0 do
  X := X + 3;
writeln(X)
```

4. The following is supposed to output all the positive odd numbers less than 10. It contains mistakes. What are they, and how can they be corrected?

```
X := 1;
while X <> 10 do
  begin{while}
    X := X + 2;
    write(X)
  end {while}
```

---

Round and round she goes,  
and where she stops nobody knows.  
*Traditional carnival barker's call*

---



---

## Terminating an Input Loop

If your program is reading in a list of values with a *while* loop, it must include some kind of mechanism to terminate the loop. If the program runs out of input, the program will stop, but if no provisions are made for the program to explicitly terminate the loop, the result will be an error condition that terminates the program abnormally. There are four commonly used methods for terminating an input loop:

1. Asking before iterating.
2. Heading the list with its size.
3. Ending the list with a sentinel value.
4. Running out of input.

We will discuss them in order.

The first method is simply to ask the user if there is more input.

```
Sum := 0;
writeln('Are there any numbers in the list? (y/n)');
readln(Ans);
while (Ans = 'y') or (Ans = 'Y') do
```

*asking before  
iterating*



```

begin{while}
  writeln('Enter number: ');
  read(Number);
  Sum := Sum + Number;
  writeln('Are there more numbers? (y/n) ');
  readln(Ans)
end {while}

```

This is sometimes acceptable and is very useful in certain situations. However, for a long list this method is very tiresome. Imagine typing in a list of 100 numbers this way. The user is likely to progress from happy to sarcastic to angry and frustrated. When you are reading in long lists, it is preferable to include only one stopping signal.

*lists headed  
by size*

On those occasions when the user naturally and easily knows the size of the list beforehand, the program can ask the user for the size of the list. This is the method we used in our sample program in Figure 7.2.

*sentinel  
value*

Perhaps the neatest way to terminate a loop that reads in a list of values is with a *sentinel value*. A sentinel value is one that is somehow distinct from all the possible values on the list and so can be used to signal the end of the list. For example, if the loop reads in a list of positive numbers, a negative number can be used as a sentinel value to indicate the end of the list. A loop such as the following can be used to add a list of nonnegative numbers:

```

Sum := 0;
read(Number);
while Number >= 0 do
  begin
    Sum := Sum + Number;
    read(Number)
  end
end

```

Notice that the last number in the list is read but is not added into Sum. To add up the numbers 1, 2, and 3, the user adds a negative number to the end of the list, like so:

1 2 3 -1

The final -1 is read in but is not added into the sum.

Also notice that when using a sentinel value, we reversed the order of summing and reading within our *while* loop. With a sentinel value, we want the loop to end as soon as the sentinel value is read, and so the loop needs to have the *read* statement at the end. We also needed to place a *read* statement before the *while* loop so that the boolean expression would be defined and so that the variable Number would have a defined value in the first assignment statement.

In order to use a sentinel value in the way just discussed, the list must be known to exclude at least one value of the data type in question. If the list consists of integers that might be any value whatsoever, then there is no value left to serve as a sentinel value. In this situation, you must resort to some other method to terminate the loop.

*running out  
of input*

As already noted, a loop that simply runs out of input will terminate with an error condition. However, if special provisions are made within the program, then in some situations it is possible to test whether there is more input and to end a loop gracefully if there is none. The next section discusses one way to do this.

## EOLN

Using the special boolean expression `eoln`, a Pascal program can detect the end of a line. This expression allows a user to mark the end of a list simply by pressing the return key. It is not very convenient for long lists, but it does work well for lists short enough to fit comfortably on one line. The identifier `eoln` is a boolean expression that tells the program when it is at the end of a line. When the program still has input available on the line that is currently being read, `eoln` has the value `false`, but when the end of a line is encountered, the value of `eoln` changes from `false` to `true`. Less formally, `eoln` is `true` when the program reaches the end of a line of input and is `false` otherwise.

Figure 7.4 shows a rewritten version of the program in Figure 7.2. This version

### Program

```
program AddUp(input, output);
{Computes the sum of a list of integers entered as input.}
var Number, Sum: integer;
begin{Program}
  writeln('Enter a list of integers all on one line');
  writeln('and then press the return key. ');
  writeln('Do not type any blanks after the last number. ');
  writeln('I will compute their sum. ');

  Sum := 0;
  while not eoln do
    begin{while}
      read(Number);
      Sum := Sum + Number
    end; {while}

  writeln('The sum is ', Sum)
end. {Program}
```

### Sample Dialogue 1

```
Enter a list of integers all on one line
and then press the return key.
Do not type any blanks after the last number.
I will compute their sum.
4 6 2
The sum is 12
```

### Sample Dialogue 2

```
Enter a list of integers all on one line
and then press the return key.
Do not type any blanks after the last number.
I will compute their sum.
4 6 2 -3
The sum is 9
```

**Figure 7.4**  
**Program using**  
**`eoln`.**

*not eoln*

uses `eoln` to detect the end of a line, rather than asking the user how many numbers there will be. The *not* before `eoln` reverses true and false, and so *not eoln* is true when the program is not at the end of the line. As illustrated in Figure 7.4, `eoln` need not be placed in parentheses when it is preceded by *not*.

*end-of-line  
character*

The exact method for determining the value of `eoln` involves a special character called the *end-of-line character*. When the user presses the return key, that sends a special character to the computer as input. This character is called the end-of-line character. The character does not appear on the screen, but its presence is indicated by the start of a new line. When the program has read all the data on a line, the next character of input is this end-of-line character. When the program executes a `readln`, this end-of-line character is skipped over. The boolean `eoln` has the value true whenever the next input character is the end-of-line character; otherwise, its value is false. Normally, one can simply think of `eoln` as being true when the computer is “at the end of a line,” but in some subtle situations you may need to think in terms of the end-of-line character.

It is possible to read the end-of-line character into a variable of type `char`, but the result is neither interesting nor typically useful. The character is converted into a blank when you do so.

---

## Pitfall

### Use of EOLN with Numeric Data

There are some special problems associated with `eoln` and numeric data. When reading numbers, many Pascal implementations assume that the line ends immediately after the last number on the line. Hence, if the user types a list of numbers and then types a blank before pressing the return key, problems are likely to ensue. Typically, the program will simply stop until the user types in an additional number. If there is any danger that the user might insert extra blanks at the end of a line, then the program should instruct the user not to do so, as illustrated in Figure 7.4.

---

## Reading from a File (Optional)

When reading a large amount of data using a loop (or any other method), it is often convenient to first enter the data into a file and then have the program read from the file. The general method for doing this with a loop is discussed in the following section. In order to use this method you must first know how to write programs that read from a file. The basic technique for doing this is explained in Part I of Chapter 13. You can read Part I of Chapter 13 without reading the material that comes before it. You can then return to this point and resume reading.

---



## Off-Line Data and a Preview of EOF (Optional)

Data is often prepared ahead of time rather than being entered by the user from the keyboard. This type of input is sometimes called *off-line data*. For example, the input might consist of experimental data collected over a long period of time and given to a program for statistical analysis. When reading off-line data, the compiler is not obtaining its data from the user, and so prompt lines such as the following are not used:

```
writeln('Enter a list of nonnegative integers');
writeln('and I will compute their sum. ');
writeln('Place a negative number after the list. ');
```

Nobody is there to read such lines and to interact with the program.

There is, however, one important way that a Pascal program can interact with off-line data in order to detect the end of the input data. Off-line data is stored in units called *files*. When data is read off-line from one of these files, the end of the file can be detected by a Pascal program in much the same way that the program can detect the end of a line. The predefined boolean expression *eof* is *false* as long as there is any input left, and it changes to *true* as soon as the program has read all the input. The use of *eof* is similar to that of *eoln*. For example, the following loop sums a list of numbers that are read off-line. The numbers are stored one per line; it is safer to use *readln* rather than *read* when using *eof*.

```
Sum := 0;
while not eof do
  begin{while}
    readln(Number);
    Sum := Sum + Number
  end {while}
```

You must somehow specify that a program is using off-line data. In Chapter 13 we discuss one method, called *text files*, that works for all Pascal systems. If you want to cover some of that material earlier, you can read in Chapter 13 now. You can read and understand that material without having read the intervening information. If you are using TURBO Pascal, you can use the material in Chapter 13, or alternatively, you can use a method called *redirection*, which is discussed in Appendix 12. In either case, you can then return to this point and continue reading.

## Modifying an Algorithm

One way to solve a problem is to modify a solution for a similar problem. Figure 7.4 gives a program to compute the sum of a list of numbers. A simple modification can change it into a program to compute the average of the list of numbers. The modification is to add a variable *Count* to the loop that will count the number of times the loop is iterated and hence will end with a value equal to the number of numbers read in. To obtain the average, the program simply divides the sum by *Count*. The modified program is given in Figure 7.5.

**Program**

```

program CompAve(input, output);
{Computes the average of a list of integers entered as input.}
var Number, Sum, Count: integer;
    Average: real;
begin{Program}
    writeln('Enter a list of integers all on one line');
    writeln('and then press the return key. ');
    writeln('I will compute their average. ');

    Sum := 0;
    Count := 0; {Counts the number of numbers.}

    while not eoln do
        begin{while}
            read(Number);
            Sum := Sum + Number;
            Count := Count + 1
        end; {while}

    if Count = 0 then
        writeln('No numbers read in. ')
    else
        begin{Count > 0}
            Average := Sum/Count;
            writeln(Count, ' numbers read in. ');
            writeln('The average is ', Average)
        end {Count > 0}
    end. {Program}

```

**Sample Dialogue**

```

Enter a list of integers all on one line
and then press the return key.
I will compute their average.
90 88 73 50 100
      5 numbers read in.
The average is 80.2000

```

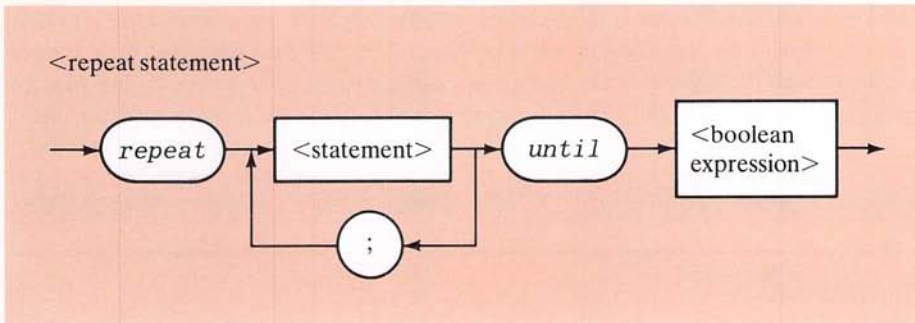
**Figure 7.5**  
Averaging  
program.

---

**The Repeat Statement**

A variant of the *while* statement is the *repeat* statement. For example, the following *repeat* statement is almost equivalent to the *while* statement in Figure 7.2. The list of statements between the identifiers *repeat* and *until* form the body of the repeat loop.

---



**Figure 7.6**  
Syntax of the  
*repeat*  
statement.

```
repeat
    read(Number);
    Sum := Sum + Number;
    LeftToRead := LeftToRead - 1
until LeftToRead <= 0
```

The syntax of the *repeat* statement is given in Figure 7.6. Notice that in a *repeat* statement the loop body consists of a list of statements and need not be a single statement. The statements are separated by semicolons, as in a compound statement. However, the syntax does not require a *begin/end* pair; the identifiers *repeat* and *until* serve a similar function.

With a *repeat* loop, the loop body is always executed at least once. When a *repeat* statement is executed, the first thing that happens is that the loop body is executed. Next, the boolean expression following the *until* is evaluated. If it evaluates to *false*, the loop body is repeated and the boolean expression is evaluated one more time. After each iteration of the loop body, the boolean expression is evaluated. If the boolean expression evaluates to *true*, the loop is ended, and the program proceeds to the next statement.

*syntax*  
of *repeat*

*action of*  
*a repeat*  
*statement*

## Comparison of the While and Repeat Loops

One syntactic difference between a *repeat* statement and a *while* statement is that the body of a *repeat* statement is a list of statements, whereas the body of a *while* statement is a single statement. However, since the body of the *while* statement can be a compound statement, this is not a significant difference.

A slightly more substantive difference is that a *while* statement terminates when the controlling boolean expression evaluates to *false*, whereas a *repeat* statement terminates when the value of the boolean expression is *true*. This is a point to be aware of if you ever translate from one type of loop to the other. Our sample *repeat* loop of the previous section has essentially the same loop body as the *while* statement it mirrors, but the boolean expression is the negation of the one occurring in the *while* statement.

The main difference between the two types of loop statements, however, is that with a *repeat* statement, the body of the loop is always executed at least once; a *while* statement is more general and allows for the possibility that the loop body may



not be executed at all. Hence, if we use the sample *repeat* statement of the previous section to replace the *while* statement in Figure 7.2, then the program will no longer be able to cope with empty lists. You cannot use a *repeat* statement unless you are certain that, under all circumstances, the loop body should be repeated at least once.

## TURBO Pascal

### Str and Val (Optional)

TURBO Pascal provides two procedures to convert back and forth between numbers and their corresponding numeral strings.

*str* converts a number to a string. It has two arguments:

*str* (*<number>*, *<string var>*)

*<number>* is a value parameter of type either integer or real. *str* converts the numeric value to the appropriate string value and places that value in the *string* variable *<string var>*. For example, the following will set the string variable *StringVar* equal to '125':

```
N := 125;
str(N, StringVar)
```

The procedure *val* accomplishes the reverse process. It has three arguments:

*val* (*<string>*, *<number var>*, *<error flag>*)

*<string>* is a value parameter of a *string* type whose value is converted to a number. That number becomes the value of *<number var>*. The variable *<number var>* may be of type either integer or real. *<error flag>* must be a variable of type integer. It is set to 0 if the conversion is possible. Otherwise, it is set equal to the position of the first character that made the conversion impossible.

*val* can be used in an input loop to recover from incorrectly typed numeric input. The input is read into a variable of a string type, and then *val* is used to check whether the input is correct and to convert it to an integer value. Possible declarations and a sample input loop are given below:

```
var StringVar: string[10];
    N, Error: integer;
. . .
repeat
  writeln('Enter an integer: ');
  readln(StringVar);
  val(StringVar, N, Error);
  if Error <> 0 then
    writeln('Not a correctly formed integer.')
until Error = 0
```

---

## Self-Test Exercises

5. What is the output of the following (when embedded in a correct program with *X* declared to be of type integer)?

```
X := 10;
repeat
  X := X - 3
until X <= 0;
writeln(X)
```

6. What is the biggest difference between a *repeat* and a *while* loop?

7. Show that any *while* statement can be replaced by a statement consisting of an *if-then* statement that contains a *repeat* statement as a subpart. Show that any *repeat* statement can be replaced by a compound statement and a *while* statement.

8. Write a program that reads in a line of text and then echoes just the last letter. The characters are read one at a time into a variable of type *char* using a loop, the loop is terminated using *eofln*, and then the last character read in is echoed. So for the input

**pineapple pasta**

the output is the single letter *a*.

---

When we see the same effect always recur, we infer a natural necessity in it, as that there will be a tomorrow, etc. But nature often deceives us, and does not subject herself to her own rules.

*Blaise Pascal*

---

---

## Case Study

---

### Testing a Procedure

We have advocated testing procedures separately by writing a small test program for each procedure. To add to our confidence in the correctness of the procedure, we need to test it with a number of different parameter values. A *repeat* loop in the test program will allow us to test the procedure as often as we wish without having to rerun the test program.

For example, in Chapter 6 we wrote a program to test a procedure named *OutputCoins*. That program tested one set of parameter values and then terminated. If we add a *repeat* loop, as shown in Figure 7.7, then we can test the input on several different sets of parameter values without having to rerun the program.

---

**Program**

```

program Test(input, output);

var TotalAmount, Quarters, Dimes, Nickels, Pennies: integer;
    Ans: char;

procedure OutputCoins(A, Q, D, N, P: integer);
{Outputs a collection of coins that total to A cents.
Precondition:  $0 < A \leq 99$  and the coins total to A,
that is  $25*Q + 10*D + 5*N + P = A$ .}

    <The rest of the procedure is shown in Figure 6.5.>

begin{TestProgram}
    repeat
        writeln('Enter: Quarters, Dimes, Nickels, Pennies: ');
        readln(Quarters, Dimes, Nickels, Pennies);
        TotalAmount := 25 * Quarters
                      + 10 * Dimes
                      + 5 * Nickels
                      + Pennies;
        OutputCoins(TotalAmount, Quarters, Dimes, Nickels, Pennies);
        writeln('Do you want to test again? (y/n) ');
        readln(Ans)
    until (Ans = 'n') or (Ans = 'N');
    writeln('End of testing')
end. {TestProgram}

```

**Sample Dialogue**

```

Enter: Quarters, Dimes, Nickels, Pennies:
1 0 1 2
32 cents can be given as:
one quarter, one nickel and 2 pennies
Do you want to test again? (y/n)
yes
Enter: Quarters, Dimes, Nickels, Pennies:
2 0 1 2
57 cents can be given as:
2 quarters, one nickel and 2 pennies
Do you want to test again? (y/n)
no
End of testing

```

**Figure 7.7**  
Testing a  
procedure.



When reading Figure 7.7, there is no need to look back at the previous chapter to see the body of the procedure. The procedure heading and the comment are all you need in order to understand the loop. The sample dialogue is short due to space limitations. This procedure should be tested on many more than the two sets of values shown.

---

## Case Study

---

### Finding the Largest and Smallest Values on a List

One often wants to end a loop partway through the body of the loop. If the loop has three statements, then on the last iteration of the loop body you might want the loop to execute only the first one or the first two statements and then end. Neither the *while* loop nor the *repeat* loop has provisions for doing this directly. However, we can usually rewrite the loop to obtain the same effect. In this case study, we will illustrate a technique for doing exactly that.

#### Problem Definition

As a sample design problem, consider the task of writing a program to determine the largest and smallest value on a list of positive numbers. We will assume that the list is not empty and that it is followed by a negative number that serves as a sentinel value marking the end of the input list.

#### Discussion

One approach is to use two variables, *Max* and *Min*, to hold the largest and smallest values read in so far. After each number is read in, it is compared to both *Max* and *Min* to see if this new number is a new low or a new high value. The basic operations in the loop body are as follows:

```
begin{loop}
  read(Next);
  if Next > Max then
    Max := Next;
  if Next < Min then
    Min := Next
end {loop}
```

Since the loop body uses the values of *Max* and *Min*, they must be initialized so that they have a value the first time through the loop. After the first number is processed, the values of both *Max* and *Min* should be equal to this number, since at that point it is both the largest and the smallest number seen so far. One way to initialize *Max* and *Min* is to process one number before starting the loop. The following, placed before the loop, will initialize the values correctly.

*ALGORITHM*  
*first version*

*initializing*  
*variables*

```

read (Next) ;
Max := Next;
Min := Next;

```

*stopping  
partway  
through the  
loop body*

At this point, we are almost through designing the algorithm and even writing the Pascal code. All we need do is design the Pascal code so that the loop stops as soon as a negative number is read in. This does present a problem because, as we have written the loop body, the read statement occurs first. We would like the program to not execute the two remaining *if-then* statements after it reads in the negative sentinel value. By coincidence, one of the statements causes no problem, because the test

```
Next > Max
```

will fail when *Next* has a negative value; it still would be neater not to execute this statement, however. Moreover, the other statement would cause the value of *Min* to be set to the negative sentinel value, and that is an incorrect value for *Min*. This is a problem because both the *while* and the *repeat* loops execute the complete loop body on each iteration. Obviously, we must make some kind of change in the loop body. One way to proceed is simply to try random rearrangements of the statements in the loop body. For small loops this trial and error method may work, but it is preferable and faster to proceed instead in a systematic way.

*possible  
loop bodies*

To understand our approach to systematically changing the loop body, you must think of the loop in terms of its action and not just in terms of how the loop body is written in Pascal. On a long list of numbers, the loop body we designed would execute the list of statements displayed in Figure 7.8. If you study that list, you will see that there is more than one way of finding repeated patterns of statements. At the opening of

```

read (Next) ;
if Next > Max then
    Max := Next;
if Next < Min then
    Min := Next;
read (Next) ;
if Next > Max then
    Max := Next;
if Next < Min then
    Min := Next;
read (Next) ;
if Next > Max then
    Max := Next;
if Next < Min then
    Min := Next;
read (Next) ;

```

**Figure 7.8**  
Program actions to  
be expressed as a  
loop.

this section, we gave one loop body obtained from the pattern observed starting with the first statement in the list. If we ignore the first read statement, however, we see a different pattern that repeats after three statements, namely

```
if Next > Max then
    Max := Next;
if Next < Min then
    Min := Next;
read(Next);
```

If we ignore the first two statements, we see the following repeated pattern:

```
if Next < Min then
    Min := Next;
read(Next);
if Next > Max then
    Max := Next;
```

This gives us three possible loop bodies to choose from. The second one (the one that ends with the read statement) ends where we want it to. If the loop ends after the read, then that last sentinel value will not be compared to either Min or Max. Hence, if we ignore the first read, we obtain a loop that ends properly:

```
while Next > 0 do
    begin{while}
        if Next > Max then
            Max := Next;
        if Next < Min then
            Min := Next;
        read(Next)
    end {while}
```

This is the correct loop, but the program design is not yet complete. We obtained this loop by ignoring that first read statement in the list of statements we wanted the program to execute. To make the program work correctly, we must include that one statement by placing it before the loop. The complete program, with the first read statement outside the loop body, is shown in Figure 7.9. The next section summarizes the design technique we used in this case study.

---

## Unrolling a Loop

The technique used in the last section is sometimes called *unrolling the loop*. When designing a loop, we can imagine that the list of statements that form the body are written on a ribbon. The list of statements is repeated a number of times to show the statements that will be executed as the loop is iterated, and the ribbon is coiled around a spool. Each revolution of the spool contains one complete loop body. The loop body starts and ends at the place at which the loose end of the ribbon leaves the spool. To change the order of the statements in the loop body, we unroll the spool of ribbon. This

---



### Program

```

program MinAndMax(input, output);
{Computes the largest and smallest values on
a list of positive integers entered as input.}
var Next, Min, Max: integer;
begin{Program}
  writeln('This program finds the largest and smallest');
  writeln('values on a list of positive integers. ');
  writeln('Enter a list of positive integers. ');
  writeln('Place a negative number at the end. ');

  read(Next);
  Max := Next;
  Min := Next;

  read(Next);
  while Next > 0 do
    begin{while}
      if Next > Max then
        Max := Next;
      if Next < Min then
        Min := Next;
      read(Next)
    end; {while}

  writeln('The largest is ', Max);
  writeln('The smallest is ', Min)

end. {Program}

```

### Sample Dialogue

```

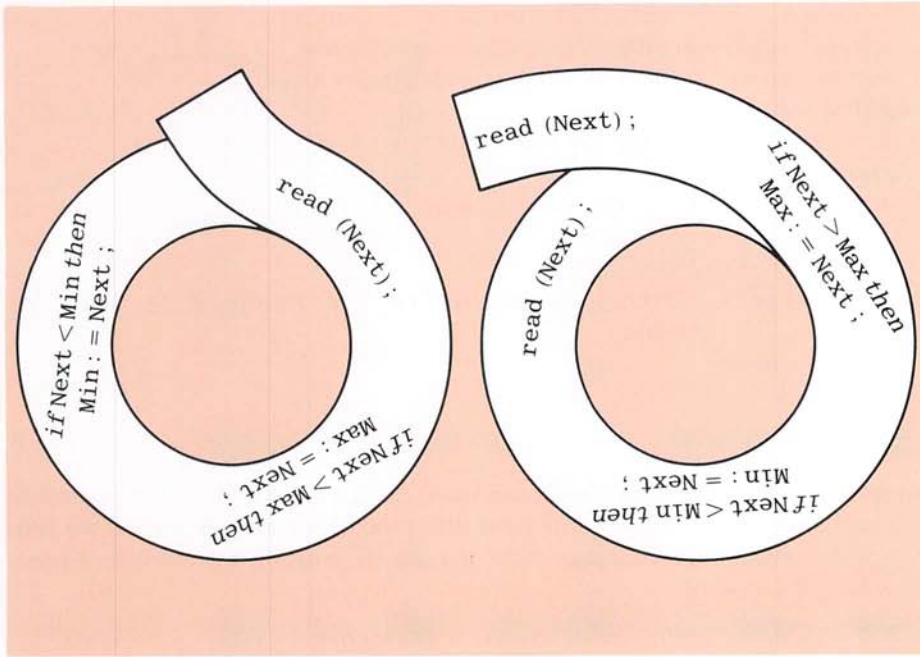
This program finds the largest and smallest
values on a list of positive integers.
Enter a list of positive integers.
Place a negative number at the end.
3 2 7 5 -1
The largest is    7
The smallest is   2

```

**Figure 7.9**  
**Program to**  
**compute largest**  
**and smallest**  
**numbers.**

process is depicted in Figure 7.10. When we unroll one or more statements, in this case the read statement, we change the list of statements that represents the loop body, and so the statements in the loop body are then in a different order. The statements that are unrolled from the spool are no longer in the loop, but they are still in the program. In the program, they are placed before the loop.

This technique is one way to design loops that we want to have start and stop at different places. We first design the loop to start where we want. After that, we unroll it until the loop body ends where we want it to. In the program, the extra statements that



**Figure 7.10**  
Unrolling a loop.

are unrolled are placed before the loop. Hence, the effect of the loop plus these extra statements is equivalent to that of starting and stopping the loop at different places in the loop body.

There is one problem that can arise when we design a loop by unrolling it. The unrolling technique assumes that the loop statements will be executed at least once. In our example we made the assumption that the list contained at least one number plus the end marker. In this case the assumption was a reasonable one. In other cases it might not be. If it is possible that the loop might not be iterated even once, then you may need to add an additional check, such as embedding a piece of code in an *if-then* statement. With any sort of loop, you should always check to see that the loop body is not repeated one too many times or one too few times.

*check  
the first  
iteration*

## Designing Robust Programs

A program that aborts or does something useless because the user has made a slight typing mistake can be maddening to use. To the extent that it is possible, you should design your programs so that they can cope with minor mistakes on the part of the user. A program should be *robust* enough to perform correctly despite some minor abuse. A program should adapt itself to the user as much as possible, rather than the other way around. Such easy-to-use programs are often referred to as *user friendly*, a pleasant term for a pleasant concept.

For example, if the user is supposed to enter a letter, then the program should

```

procedure ReadCheck(var Amount: integer);
{Reads in a value in the range 0 to 100 from the keyboard.
Allows the user to retry until the value is in the correct range.}
begin{ReadCheck}
    writeln('Enter an amount in the range 0 to 100. ');
    readln(Amount);
    while (Amount < 0) or (Amount > 100) do
        begin{Retry}
            writeln('Try again. ');
            writeln('Enter an amount in the range 0 to 100. ');
            readln(Amount)
        end {Retry}
    end; {ReadCheck}

```

**Figure 7.11**  
Robust input  
procedure.

*upper- and lower-  
case letters  
in input*

accept either an uppercase or a lowercase letter. If the letter is used to control a *case* statement, then both upper- and lowercase letters should normally be used on the label lists. If the user is supposed to type in “y” for yes, the program test should read something like the following:

```
if (Ans = 'y') or (Ans = 'Y') then . . .
```

We have already used these techniques in the programs in Figures 6.12 and 7.7. If the idea is not clear, you may want to review those programs.

*retry  
input*

Another helpful feature is to echo input and let the user retype the data if it has been entered incorrectly. Sometimes the program can check the data for appropriateness. The procedure in Figure 7.11 reads in an integer and allows the user to retry until it is within the desired range.

---

## Invariant Assertions and Variant Expressions (Optional)

A program with loops can be more complicated and more difficult to understand than the simple programs we examined prior to this chapter. Consequently, you need to document the loops in your programs carefully. In this section we will describe some widely used methods for commenting loops.

Recall that an assertion is a comment that states something that the programmer expects to be true whenever the program execution reaches the assertion. If the program is correct, then the assertion will be true. The only sort of specialized assertions we have had any significant contact with are preconditions and postconditions. As with other programming constructs, loops of any complexity should be documented with preconditions and postconditions. The precondition asserts what is true before the loop is executed, and the postcondition asserts what should be true after the loop is executed.

*invariants*

There is one other kind of assertion that is frequently used to document loops. This type of assertion is called a *loop invariant* or simply an *invariant*. An invariant is an

---



assertion that is true before the loop is executed and that remains true after each iteration of the loop. Thus, an invariant is true before the loop is executed, after it is executed, and at certain specific times while the loop is in progress. As a simple example of a loop, consider the following piece of code:

```
N := 10;
Sum := 10;
while N > 1 do
  begin{while}
    N := N - 1;
    Sum := Sum + N
  end {while}
```

Since an invariant is an assertion that is true before the loop is executed, it must be true just before the program execution gets to the word *while*. It must also be true after each iteration of the loop; hence, it must be true just before the word *end*. This implies that it is true each time the program reaches the *while* loop. It also implies that it will be true after the *while* loop has ended. One invariant of this loop is the assertion

*$N \geq 1$  and the value of Sum is the sum of  
all the integers  $x$  such that  $N \leq x \leq 10$ .*

The assertion is true before the loop is executed, because then the values of Sum and N are both 10. Each iteration of the loop decreases N by one and then increases the value of Sum by the new value of N. Hence, after each iteration the value of Sum is the sum of the integers from the new value of N up to 10. The loop is not iterated unless N has a value greater than one. After each iteration, therefore, we know that  $N \geq 1$ , as stated in the invariant assertion. The values of N and Sum are changed, but they are changed in such a way that the assertion is true after each loop iteration. This is why it is called an “invariant”; its truth does not vary from one iteration of the loop to the next.

A correct invariant for a loop will be true after each iteration of the loop. However, it may become false at some time during the execution of the loop. In fact, any useful invariant will become false at some time during the execution of the loop. The loop shown in Figure 7.12 has been annotated to indicate when the invariant is true and when it is false. (The comment that starts out “*The variant expression . . .*” will be explained shortly; for the moment you can safely ignore it.)

There is always more than one invariant for a loop. For example,  $\text{Sum} \geq 10$  and  $N \geq 0$  is also an invariant for the loop in Figure 7.12, but a much less informative one. The invariant chosen for documenting the loop should say something informative about what you want the loop to do. A good invariant serves as a bridge between the precondition and the postcondition. It says something about how the precondition is transformed into the postcondition.

As we have described invariants, it sounds as if the precondition must imply the invariant. This is not quite true. Most loops require that certain variables or other items be initialized before the loop is executed. These initialization statements may derive from unrolling the loop or from other considerations. In any event, the initialization statements are conceptually attached to the loop. Hence, the precondition goes before

```

N := 10;
{The value of N is 10.}
Sum := 10;
{N >= 1 and the value of Sum is the sum of
all the integers x such that N <= x <= 10.}
{The variant expression N is decreased until it is
less than or equal to the threshold of one.}
while N > 1 do
  begin{while}
    N := N - 1;
    Sum := Sum + N
    {N <= 1 and the value of Sum is the sum of
all the integers x such that N <= x <= 10.}
  end {while}
{The value of Sum is the sum of all integers from 1 to 10.}

```

**Figure 7.12**  
(Optional)

A fully  
documented loop.

variant  
expression

the initialization statements. The invariant goes after any initialization statements. For this reason, the invariant need not follow from the precondition alone.

One way to see that a loop will eventually end is to find some quantity that is changed in each iteration of the loop until it reaches or passes some given value. An expression for such a quantity that changes each time the loop is executed is called a *variant expression*. The value that the variant expression must reach or pass is called a *threshold*. There are two important properties that the variant expression and threshold must have:

1. There must be some fixed amount such that the value of the variant expression decreases by at least this amount each time the body of the loop is executed.
2. Whenever the value of the variant expression is less than or equal to the threshold, the loop must terminate.

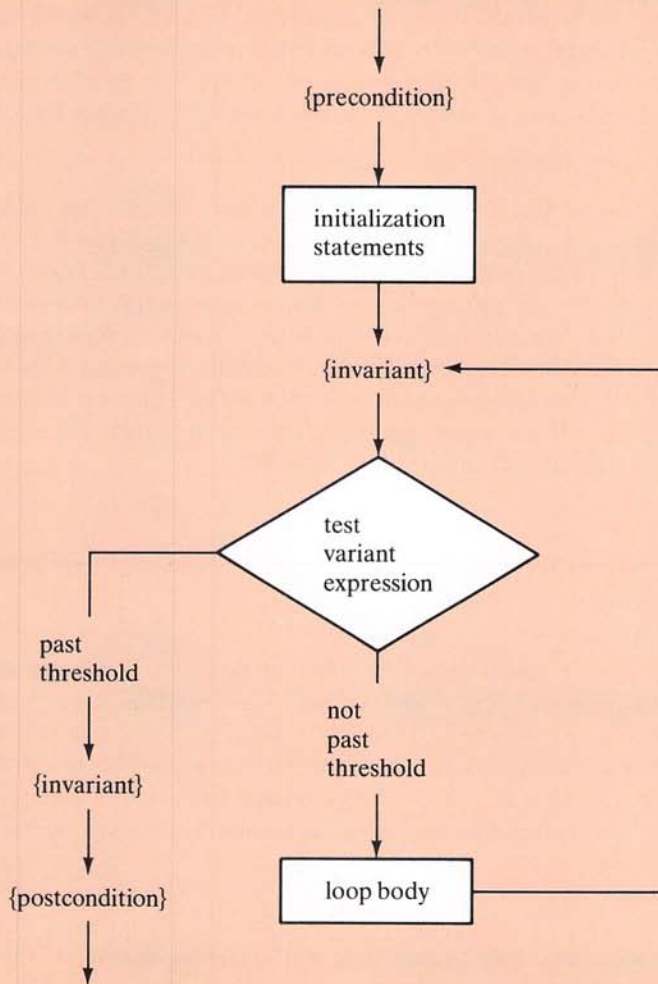
Any loop that has a variant expression and threshold with these properties will eventually terminate. If the value of the expression is below the threshold, the loop terminates right away. In the more typical case, in which the initial value of the variant expression is greater than the threshold, we know that the value will be decreased each time the loop is executed and so will eventually fall below the threshold. We also know that the loop will terminate when that happens. The process is shown diagrammatically in Figure 7.13.

A variant expression for our sample *while* loop is the simple expression *N*. It is decreased by one on each iteration, and once it is less than or equal to the threshold of one, the loop must terminate.

increasing to  
a threshold

We chose to define variant expressions and thresholds in terms of a decreasing quantity. It is also possible to think in terms of an increasing quantity. In such cases, the loop terminates when the variant expression is greater than the threshold.

Invariant assertions and variant expressions complement each other very well. One tells what remains constant and the other what changes in a loop. If they are chosen carefully, the postcondition will follow from the invariant assertion and the fact



**Figure 7.13**  
(Optional)  
**Invariant assertion**  
**and variant**  
**expression.**

that the variant expression has passed the threshold. This means that if these items are chosen carefully, they can be used to demonstrate conclusively that a loop behaves as the programmer claims it does.

For example, consider the loop shown in Figure 7.12. The invariant assertion says that the following will be true after each iteration of the loop:

*$\{N \geq 1 \text{ and the value of Sum is the sum of all the integers } x \text{ such that } N \leq x \leq 10.\}$*

*demonstrating  
a postcondition*



Since it is true after each loop iteration, it must be true when the loop ends, and so  $N$  is greater than or equal to one when the loop ends. Because the *while* statement does not terminate unless  $N$  is less than or equal to one, we therefore know that when the loop ends, the variant expression  $N$  is both greater than or equal to one and less than or equal to one. This means that the value of  $N$  is one. Since  $N$  is equal to one and the invariant assertion holds, we see that the postcondition must hold when the loop ends:

*The value of Sum is the sum of all integers from 1 to 10.*

It may seem that all this documentation is overdoing things a bit. Indeed, the simple loop in Figure 7.12 is buried in comments, some of which can just as well be omitted. Normally, it is sufficient to include the invariant assertion only once, typically at the end of the loop. The precondition stating that the value of  $N$  is 10 is obvious and should not be included. We only included it in order to have a complete and simple example. You should have a variant expression and a threshold in mind when writing a loop. However, if they are obvious enough, you need not include them in a comment. For complicated loops all this documentation should be included. For very simple loops, omitting some details can actually aid readability.

---

## For Statements

The *while* and *repeat* statements are the only loop mechanisms that you absolutely need. In fact, the *while* statement alone is enough. However, there is one sort of loop that is so common that Pascal includes a special “tailor-made” loop statement for it. In performing numeric calculations, it is common to perform a calculation with the number 1, then with 2, then with 3, and so forth until some last value is reached. For example, to write out the multiplication table for the number 2, we want the computer to perform the following:

```
writeln('2 times ', I, ' is ', 2*I)
```

first with  $I$  equal to 1, then with  $I$  equal to 2, and so forth up to, say, 9. One way to do this is with a *while* statement, such as

```
I := 1;
while I <= 9 do
begin
  writeln('2 times ', I, ' is ', 2*I);
  I := I + 1
end
```

Although a *while* loop will do here, this sort of situation is just what the *for* statement was designed for. The above piece of code will produce the same output as the following *for* statement:

```
for I := 1 to 9 do
  writeln('2 times ', I, ' is ', 2*I)
```

---

*for* statements come in two very similar varieties. The above example is of the first variety whose general form is

```
for <control variable> := <initial exp> to <final exp> do
    <body>
```

The <body> may be any single Pascal statement. In particular, it can be a compound statement. For now, we will insist that the *control variable* be a variable of type *integer*. Therefore, the loop control variable must be declared to be of type *integer*. Eventually we will see how to use other types for the control variable, but the idea is clearest in the case of *integer* variables. The expressions <initial exp> and <final exp> must evaluate to values of type *integer*. Their values are called the *initial value* and *final value*, respectively, of the loop control variable. Typically, we want the initial value to be less than the final value, but this is not required. Typically, <body> includes some reference to the <control variable>, but this also is not required.

When the *for* loop is executed, the expressions <initial exp> and <final exp> are evaluated to obtain the initial and final values for the loop control variable. After that, <body> is executed, first with <control variable> equal to the initial value, then with <control variable> equal to the initial value plus 1, then with <control variable> equal to the initial value plus 2, and so on, increasing the value of <control variable> by one each time. The last time through the loop, <body> is executed with <control variable> equal to the value that was obtained from the expression <final exp>. This behavior is diagrammed in Figure 7.14.

A *for* loop need not start with the number one. For example, the following will give the two multiplication table for all values from -9 up to +9:

```
for I := -9 to 9 do
    writeln('2 times ', I, ' is ', 2*I)
```

The loop control variable is automatically increased by one each time through. There is no need to include anything like the following in the body of the loop:

```
I := I + 1
```

In fact, it is an error to do so. The body of a *for* loop is not allowed to change the loop control variable in any way. If it does, the effect of the *for* loop is unpredictable. The body can use the value of the control variable, but it cannot change it.

A loop control variable is intended to be used only to control the *for* loop and nothing else. The value it has before the *for* loop is executed is irrelevant to the *for* loop. In standard Pascal, its value is undefined after the loop terminates, which means that its value is completely unpredictable. (In *TURBO Pascal*, the value of the loop control variable when the loop terminates is its final value, provided that the loop was executed at least once. If the loop was not executed at all, its value is undefined.) It is natural to make the loop control variable a local variable. Moreover, in many Pascal implementations, the loop control variable is required to be a local variable. If a *for* loop appears in a procedure, then the loop control variable must be declared in that procedure.

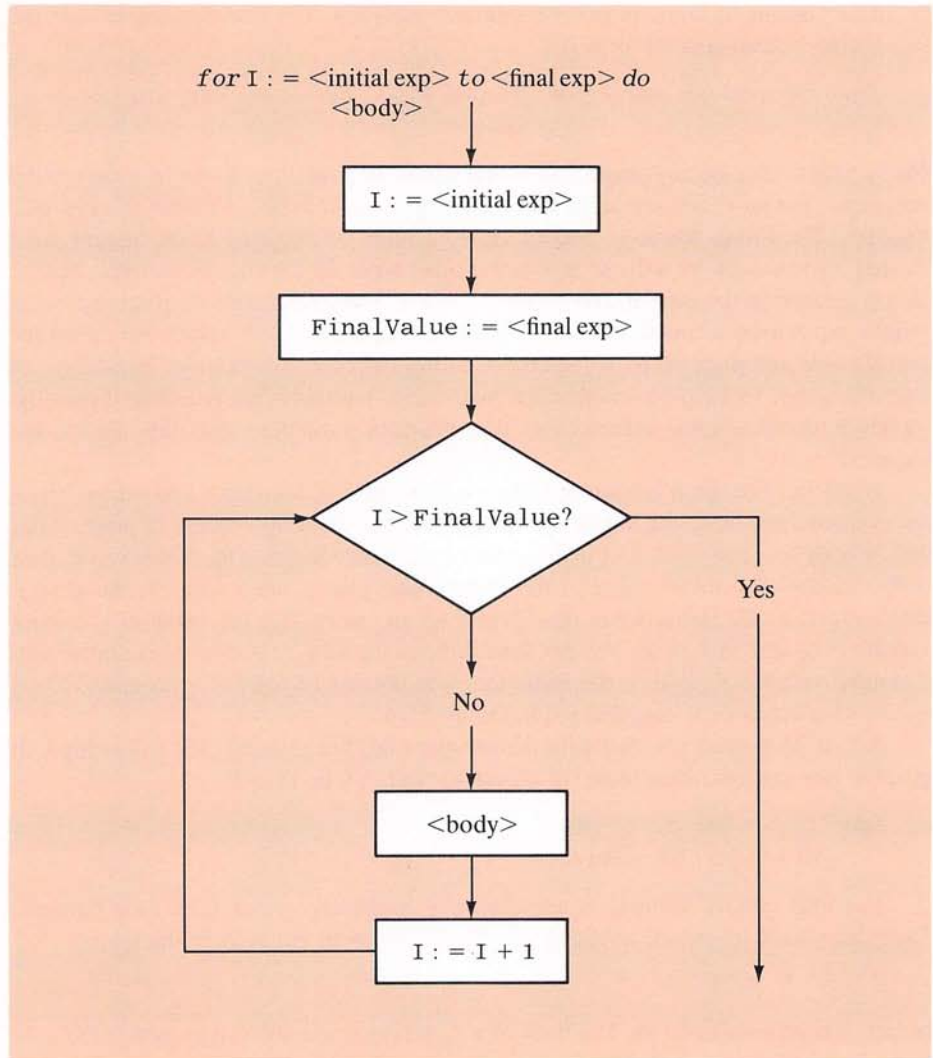
It is natural to wonder what happens when the value of <initial exp> is greater

*syntax*

*control  
variable*

*behavior  
of for  
loops*

*restrictions  
on loop  
control  
variables*



**Figure 7.14**  
Behavior of one  
type of *for* loop.

than that of <final exp>. In Pascal, when this happens the loop body is not executed at all and the program goes on to the next statement. This means that in Pascal it is impossible to write a *for* loop that is an infinite loop.

The body of a *for* statement cannot change the number of times it is iterated by changing the value of <final exp>. The technical explanation of what happens is as follows: Both <initial exp> and <final exp> are evaluated once at the start of the loop, and these two values are used to control the loop.

One cultural point that is of some significance to the design of readable programs has to do with names of *for* loop control variables. Most of our *for* loops use I and J as loop control variables. For historical reasons, it has become customary to use I,



J, and K as loop control variables. The letters carry no mnemonic value, but the custom is so well ingrained that it is pointless to fight it. When you see these letters in programs, they will very likely be loop control variables, and this convention can be a slight aid in reading the programs. Of course, if you can come up with good mnemonic names of *for* loop control variables, that is even better.

The second version of the *for* statement is very similar to the first. The only difference in syntax is that *to* is replaced by *downto*. The difference in what happens when it is executed is that with the *downto* version, the loop control variable is decreased by one each time through. In this case, the value of <initial exp> is typically greater than or equal to that of <final exp>; if it is not, the body is not executed at all, and the program proceeds to the next statement. The word *downto* is a single identifier and should not be written as two words.

In Pascal a *for* loop control variable is always changed by either plus one or minus one. Other programming languages allow it to change by any specified amount. At first this looks like a real limitation in Pascal, but it is easy to program around this limitation. To output the even numbers 0, 2, 4, 6, 8, and 10, the following trick works:

```
for I := 0 to 5 do
    write(2*I)
```

This same sort of trick can be used to get the equivalent of increments of fractional size as well. To see the effect of small changes in interest rates, one might use a *for* loop such as the following:

```
for J := 10 to 200 do
begin{for loop body}
    Rate := 0.001*J;
    Interest := Amount * Rate;
    writeln('a rate of ', Rate*100, ' percent, ');
    writeln('yields ', Interest, ' in interest')
end {for loop body}
```

*downto*

*other  
increment  
sizes*

---

## Example—Summing a Series

Figure 7.15 consists of a complete program that includes a *for* loop. The program computes the average number of times you need to flip a coin in order to get the coin to come up heads. The program uses the following formula to approximate the average:

$$\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{N}{2^N}$$

We need not concern ourselves with how the formula is derived. We will simply assume that it works. Like many such formulas, it yields an approximation to the desired value rather than the exact value. The larger the value of *N*, the better the approximation of the true value of this average. A value of 100 for *N* is more than large enough to give as much accuracy as most computers are capable of delivering, and so the program uses the formula with *N* set equal to 100.

---

```

program CoinToss(input, output);
{Computes the expected number of coin tosses needed
to obtain a head when flipping a balanced coin.}
const NumberOfTerms = 100;
var I: integer;
    Sum, Power: real;
begin{CoinToss}
    Sum := 0;
    Power := 1;
    for I := 1 to NumberOfTerms do
        begin{for}
            Power := Power * 2;
            Sum := Sum + I/Power
            {Sum is the sum of all numbers of the form:
            x/(2 to the power x) where 1 <= x <= I.
            Power is 2 to the power I.}
        end; {for}
    writeln('On the average it will take ', Sum);
    writeln('tosses of a balanced coin');
    writeln('to get heads the first time.')
end. {CoinToss}

```

**Figure 7.15**  
Summing a series.

*sums  
versus  
products*

Notice that Sum is initialized to zero, whereas Power is initialized to one. This is because Sum will contain a sum of numbers, while Power will contain a product of numbers. One way to test that these are the correct initial values is to check to see that they make the loop assertion true after one iteration of the loop.

Also notice that Power is declared to be of type real, even though its value is conceptually an integer. The reason for this is that the value of Power becomes extremely large, and on most systems would produce integer overflow, that is, its value would become larger than maxint. Since computers can store much larger real values than they can integer values, the change to type real overcomes this problem. When performing numeric calculations with *for* loops, you may encounter this often. We will have more to say about this problem in Chapter 15.

---

## Repeat N Times

One often encounters a section of pseudocode similar to the following:

```

repeat the following loop body N times
    begin{loop body}
        read (Number) ;
        Sum := Sum + Number
    end {loop body}

```

---

Some programming languages have a loop construct that corresponds exactly to this type of loop. Pascal does not, but it is easy to construct the equivalent of “repeat the loop body *N* times.” Our first example of a program with a loop, shown in Figure 7.2, illustrates one way to do this with a *while* loop, but the easiest way to implement it is with a *for* loop:

```
for I := 1 to N do
  begin{for}
    read (Number);
    Sum := Sum + Number
  end {for}
```

At first this may seem strange, since the loop control variable (in this case *I*) does not appear in the loop body. There is nothing wrong with that. It is perfectly legitimate to use this variable for nothing other than counting the number of loop iterations.

---

## Invariant Assertions and For Loops

(Optional)

*for* loops do not lend themselves to documentation with invariants as easily as *repeat* and *while* loops do. This is because the natural statement of an invariant for a *for* loop usually includes a reference to the loop control variable. This presents a problem, since the loop control variable of a *for* loop is frequently undefined or irrelevant both before the loop is executed and also after the *for* loop is executed. The problem is more of a notational one than a conceptual one. When you are designing a *for* loop, it makes sense to think in terms of invariants. However, when you are actually writing the *for* loop comments, it is usually easier and clearer to use some close approximation to an invariant. If the *for* loop body is a compound statement, one possible solution is to include an assertion within that compound statement. Since the loop control variable is well defined within the body, that assertion can refer to the loop control variable in a clearly meaningful way.

There is no need to design variant expressions and thresholds for *for* loops, because a *for* loop already has a built-in variant expression and a built-in threshold. The variant expression is the loop control variable, and the threshold is its final value.

---

## What Kind of Loop to Use

When you are designing a loop, the choice of which type of Pascal statement to use is best postponed to the end of the design process. First, design the loop using pseudocode, and then translate the pseudocode into Pascal code. At that point it is easy to decide what type of Pascal loop statement to use.

If the loop involves a numeric calculation that is repeated a fixed number of times using a value that is changed by equal amounts each time through the loop, then use a *for* loop. In fact, any time you have a loop for a numeric calculation, you should consider using a *for* loop. It will not always be possible, but in many cases involving

---



numeric calculations it is the clearest and easiest loop to use. A *for* loop is also the easiest way to construct the equivalent of an instruction that says “repeat the loop body N times.”

In all other cases, you must use a *repeat* loop or a *while* loop. It is fairly easy to decide which one to use. If you want to insist that the loop body be executed at least once, then you can use a *repeat* loop. If there are circumstances under which the loop body should not be executed at all, then you cannot use a *repeat* loop and so must use a *while* loop. A common situation that demands a *while* loop is one in which there is a possibility of no data at all. For example, a program that reads in a list of exam scores may run across students who have taken no exams, and hence the input loop may be faced with an empty list. This calls for a *while* loop.

---

## Nested Loops

Figure 7.16 contains a program that writes out the multiplication table. The *for* loop in the main body of the program writes out one line for each value of N from 0 to 9 using the procedure call `PrintRow(N)`. This procedure call writes out the number N to label the row and then writes the table entries for N times the numbers 0 to 9.

The body of a loop may contain any kind of statement, and so it is possible to have loops nested within loops. The program in Figure 7.16 contains a loop within a loop. Normally, we do not think of it as containing a nested loop, because the inner loop is contained within a procedure and it is the procedure call that is contained in the outer loop. There is a lesson to be learned from these unobtrusively nested loops, namely that nested loops are no different from any other loops. The program in Figure 7.17 is rewritten with the nested loop explicitly displayed. The nested loop in Figure 7.17 is iterated once for each value of N from 0 to 9. For each such iteration, there is one complete execution of the inner *for* loop.

The two versions of our multiplication table program are equivalent, but many people find the version in Figure 7.16 easier to understand because the loop body is a procedure. This is an example of procedural abstraction. When considering the outer loop, one thinks of printing a row as a single operation and not as a loop. When writing or reading explicitly displayed nested loops like the one in Figure 7.17, one should think of the loop body as a unit in this way whether or not it is packaged into a procedure.

---

## Debugging Loops

No matter how carefully a program is designed, mistakes will still sometimes occur. In the case of loops, there is a pattern to the kinds of mistakes most often made. Most loop errors involve the first or last iteration of the loop. If you find that your loop does not perform as expected, check to see if the loop is iterated one too many or one too few times. Iterating the loop one too many or one too few times is one of the most common loop bugs. Be sure that you are not confusing “less than” with “less than or equal to.” Be sure that you have initialized the loop correctly. Check the possibility that

*procedural  
abstraction*

*common  
errors*

*off-by-one  
errors*

**Program**

```

program TimesTable(input, output);
{Writes out the multiplication table.}
const Width = 4; {Space to hold one number entry in table.}
var N: integer;

procedure PrintTop;
{Prints a heading for the multiplication table
and a line of digits from 0 to 9.}
var I: integer;
begin{PrintTop}
  writeln('The Multiplication Table' :30);
  write('*': Width);
  for I := 0 to 9 do
    write(I:Width);
  writeln
end; {PrintTop}

procedure PrintRow(N: integer);
{Writes out N followed by N*0, N*1, N*2, . . . , N*9}
var M: integer;
begin{PrintRow}
  write(N:Width); {Specifies first factor}
  for M := 0 to 9 do
    write(N * M:Width);
  writeln
end; {PrintRow}

begin{Program}
  PrintTop;
  for N := 0 to 9 do
    PrintRow(N)
end. {Program}

```

**Output**

```

      The Multiplication Table
*  0  1  2  3  4  5  6  7  8  9
0  0  0  0  0  0  0  0  0  0
1  0  1  2  3  4  5  6  7  8  9
2  0  2  4  6  8 10 12 14 16 18
3  0  3  6  9 12 15 18 21 24 27
4  0  4  8 12 16 20 24 28 32 36
5  0  5 10 15 20 25 30 35 40 45
6  0  6 12 18 24 30 36 42 48 54
7  0  7 14 21 28 35 42 49 56 63
8  0  8 16 24 32 40 48 56 64 72
9  0  9 18 27 36 45 54 63 72 81

```

**Figure 7.16**  
Times table  
program, one  
version.

```

program TimesTable(input, output);
{Writes out the multiplication table.}
const Width = 4; {Space to hold one number entry in table.}
var N, M: integer;

procedure PrintTop;
{Prints a heading for the multiplication table and a line of digits from 0 to 9.}
var I: integer;
begin{PrintTop}
    writeln('The Multiplication Table' :30);
    write('*': Width);
    for I := 0 to 9 do
        write(I:Width);
    writeln
end; {PrintTop}

begin{Program}
    PrintTop;
    for N := 0 to 9 do
        begin{Row}
            {Writes out N followed by N*0, N*1, N*2, . . . , N*9}
            write(N:Width); {Specifies first factor.}
            for M := 0 to 9 do
                write(N * M :Width)
            writeln
        end {Row}
    end. {Program}

```

**Figure 7.17**  
Nested **for** loops.

the loop may sometimes need to be iterated zero times, and make sure that your loop handles that possibility correctly.

*infinite  
loops*

Infinite loops usually result from a mistake in the boolean expression of a *repeat* or *while* loop. Check to see that you have not reversed an inequality, confusing “less than” with “greater than.” Terminating a loop with a test for equality rather than with one involving a greater than or less than comparison is another common source of infinite loops.

If you check and recheck your loop and can find no error, but the program still misbehaves, then you will need to do some more sophisticated testing. First of all, make sure that the mistake is indeed in the loop. Just because the program is performing incorrectly does not mean the bug is where you think it is. If your program was designed to be modular and is divided into procedures, then it should be easy to find the approximate location of the bug or bugs. Once you have decided that the bug is in a particular loop, you should trace the key variables in the loop.

*tracing*

*Tracing* was introduced in Chapter 3. It consists of adding some extra write statements to output intermediate results, so that you can watch the program working. For example, consider the following piece of code:



```

N := 10;
Sum := 10;
while N > 1 do
  begin{while}
    Sum := Sum + N;
    N := N - 1
    {The value of Sum is the sum of all the integers y such that: N <= y <= 10.}
  end; {while}
{The value of Sum is the sum of all integers from 1 to 10.}

```

The last comment explains what the value of Sum is supposed to be. If you run this code, you will find that it does not set Sum to that value. The value of Sum will be larger than it is supposed to be. The loop can be tested by tracing the variable Sum—that is, by writing out its value after each iteration of the loop. Suitable `writeln` statements to do the job are shown in Figure 7.18.

If you embed the loop containing the trace statement in a program and run it, the source of the error will immediately become apparent. The first two values of Sum will be 10 and 20, but the value of Sum after one iteration of the loop should be  $10 + 9$  or 19, rather than 20. Thus, we can immediately see that the value of N is the source of the problem. After we discover this, it is easy to see that N should be added into Sum after it is decremented, rather than before. In other words, the correct order for the statements in the loop body is

```

N := N - 1;
Sum := Sum + N

```

Before leaving this example, we should comment on some peculiarities of the trace statements in Figure 7.18. First of all, the trace statements in the figure seem to have semicolons in strange places. This is not necessary, but it is a good idea. If we precede each trace statement with a semicolon, then we know that it will be separated from the previous statement by a semicolon. The `writeln` that was inserted just before the `end` needs the semicolons. The semicolons preceding the other trace state-

```

N := 10;
Sum := 10;
{TEMP}; writeln('N = ', N, ' Sum = ', Sum);
{TEMP}; writeln('Before loop');
while N > 1 do
  begin{while}
    Sum := Sum + N;
    N := N - 1
    {The value of Sum is the sum of all the integers y such that: N <= y <= 10.}
  {TEMP}; writeln('N = ', N, ' Sum = ', Sum);
  end; {while}
{TEMP}; writeln('After loop');
{TEMP}; writeln('N = ', N, ' Sum = ', Sum);
{The value of Sum is the sum of all integers from 1 to 10.}

```

**Figure 7.18**  
**Tracing a loop.**

ments do no harm. This saves us from the worry of having to fix up semicolons. It looks strange, but remember that the trace statements are only temporary. They and the extra semicolons will not appear in the final program.

It also may seem that the trace statements that precede and follow the loop are not needed. In this case they were not, but remember that you are using trace statements because you do not know where the error is. It is dangerous to leave any possibility unchecked. A good practice is to place one trace statement before the loop, one after the loop, and one or more inside the body of the loop, even if some of these traces are “clearly” redundant.

*long  
loops*

The idea of tracing the loop through each iteration will work fine on a loop that is iterated 10 times, but a loop that is iterated 1000 times is likely to produce too much information to digest. For example, the following will produce nothing but a blur of light on a video display screen:

```
for K := 0 to 1000 do
  begin
    Sum := Sum + K
    {TEMP}; writeln(K, Sum);
  end
```

A better approach is to take a smaller sample. One way of doing this is as follows:

```
for K := 0 to 1000 do
  begin
    Sum := Sum + K
    {TEMP}; if (K mod 100) = 0 then writeln(K, Sum);
  end
```

This will output a trace statement every time K is a multiple of 100. There is a possibility, however, that such a uniform sampling technique will fail to detect certain kinds of periodic problems. If a trace like the one above yields no insights, try a different sampling technique, such as using

```
(K mod 100) = 1
```

This will also sample every one hundredth iteration, but the value of K will not be a “round” number. If that fails, change 100 to some other number.

---

## Case Study

---

### Calendar Display

#### Problem Definition

We wish to design a procedure that outputs the traditional calendar display of a month. To simplify the problem, we will assume that the user tells the program how many days there are in the month and on what day of the week the month starts. The day of the week will be input as an integer: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so

---

forth. For example, the input values 30 (for the number of days) and 3 (for Wednesday) should produce the following output:

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

## Discussion

One approach to this problem is the following outline of an algorithm:

Output a heading with the names Sun, Mon, Tues, etc.

Output blanks up to the first day of the month.

Output the rest of the first week.

For each remaining week do the following:

*begin{week}*

write the numbers for the week;

writeln;

*end {week}*

A straightforward refinement of the above algorithm would produce the following pseudocode:

Output a heading with the names Sun, Mon, Tues, etc.

*for* I := 0 *to* (FirstDay - 1) *do*

write a blank;

*for* I := 1 *to* (7 - FirstDay) *do*

write(I);

writeln;

DayNumber := 7 - FirstDay;

*repeat*

*for* I := 0 *to* 6 *do*

write(DayNumber + I);

writeln;

DayNumber := DayNumber + 7

*until* DayNumber > (the number of days in the month)

Having started to write code for our program, we can observe a number of things wrong with our approach. First of all, the pseudocode does not work. The second 7 should be an 8, or else one number will be output twice. Fixing that will make the code look strange but will eliminate one problem. However, even after that “patch,” the output will not be correct. Unless the last week happens to end on a Saturday, that week will be output incorrectly. Another “patch” can make it work, but that will make the code even more complicated and even less understandable. The real problems are that the code is unclear and is too complicated for the task it is accomplishing. In a situation like this, it is best to try starting over.

*false  
starts*



*rethinking  
the problem*

Let us look at the task in a different way: All we want the program to do is to output the numbers 1 through the number of the last day and to insert `writeln` at the end of each week. The first of these tasks is trivial:

```
for DayCount := 1 to (the number of the last day) do
  write (DayCount)
```

To accomplish the second task, we can add a test for the end of the week inside of the `for` loop:

```
for DayCount := 1 to (the number of the last day) do
  begin{for}
    write (DayCount);
    if "end of a week" then
      writeln
    end {for}
```

To complete our second (and successful) attempt at designing our program, we need only design a test for the end of a week. A first, careless guess might be

```
if DayCount mod 7 = 0 then
  writeln
```

This will work provided the month starts on a Sunday, but again look back at our sample of what we want for output. The first week of that month starts on a Wednesday, and so it should have a `writeln` after the first four days. If started on a Wednesday, the above test will produce output that begins

Sun	Mon	Tue	Wed	Thu	Fri	Sat				
			1	2	3	4	5	6	7	
8	9	10	11	12	13	14				

**ALGORITHM**

To determine when the `writeln` should be executed, the program must count the blanks. To do this we will use another variable called `CountAll` that starts counting on Sunday even if the month starts on some other day. The correct pseudocode is as follows:

Output a heading with the names Sun, Mon, Tues, etc.

Output the initial blanks.

`CountAll := FirstDay; {for the initial blanks.}`

`for DayCount := 1 to (the number of the last day) do`

`begin{for}`

`CountAll := CountAll + 1;`

`{If (CountAll mod 7) = 0, then day number DayCount is a Saturday.}`

`write (DayCount);`

`if (CountAll mod 7) = 0 then`

`writeln;`

`end; {for}`

`writeln`

The complete procedure, embedded in a test program, is shown in Figure 7.19.

### Program

```

program Calendar(input, output);
{Displays the calendar layout of any month.}
var FirstDay, NumberOfDays: integer;
    Ans: char;

procedure DisplayMonth(NumberOfDays, FirstDay: integer);
{Displays the usual layout for a month with NumberOfDays days in it.
FirstDay codes the first day of the month: 0 for Sunday, 1 for Monday, etc.}
const Width = 4; {Field width for one day of the calendar.}
    Blank = ' ';
var DayCount, CountAll: integer;
begin{DisplayMonth}
    writeln('Sun':Width, 'Mon':Width, 'Tue':Width,
            'Wed':Width, 'Thu':Width, 'Fri':Width, 'Sat':Width);
    for CountAll := 0 to FirstDay - 1 do
        write(Blank :Width);
    CountAll := FirstDay; {for the initial blanks.}
    for DayCount := 1 to NumberOfDays do
        begin{for}
            CountAll := CountAll + 1;
            {If (CountAll mod 7) = 0, then day number DayCount is a Saturday.}
            write(DayCount :Width);
            if (CountAll mod 7) = 0 then
                writeln;
        end; {for}
    writeln
end; {DisplayMonth}

begin{Program}
    writeln('I will display the calendar of a month. ');
    repeat
        writeln('Enter the number of days in the month: ');
        readln(NumberOfDays);
        writeln('Enter the first day of month, 0 for Sunday, ');
        writeln('1 for Monday, 2 for Tuesday, and so on. ');
        readln(FirstDay);
        DisplayMonth(NumberOfDays, FirstDay);
        writeln('Do you want to see another month? ');
        writeln(' (yes or no): ');
        readln(Ans)
    until (Ans = 'N') or (Ans = 'n');
    writeln('Have a good month! ')
end. {Program}

```

**Figure 7.19**  
Calendar program.

### Sample Dialogue

```

I will display the calendar of a month.
Enter the number of days in the month:
31
Enter the first day of month, 0 for Sunday,
1 for Monday, 2 for Tuesday, and so on.
2
Sun    Mon    Tue    Wed    Thu    Fri    Sat
      6      7      8      9     10     11     12
13     14     15     16     17     18     19
20     21     22     23     24     25     26
27     28     29     30     31
Do you want to see another month?
(yes or no):
no
Have a good month!

```

**Figure 7.19**  
(continued)

---

## Starting Over

The previous case study illustrates an important engineering design principle. If a program or algorithm is very difficult to understand or performs very poorly, do not try to fix it; instead, throw it away and start over. This will result in a program that is clearer to read and that is less likely to contain hidden errors. What may not be so obvious is that by throwing out the poorly designed code and starting over, you will produce a working program faster than if you tried to repair the old code (or old pseudocode). It may seem like wasted effort to throw out the code that you worked so hard on, but that is the most efficient way to proceed. The work that went into the discarded code is not wasted. The lessons you learned by writing it will help you to design a better program and to do so faster than if you started with no experience. The code itself is unlikely help at all.

---

I beheld the wretch—the miserable monster whom I had created.

*Mary Wollstonecraft Shelley, Frankenstein*

Care is no cure, but rather corrosive,  
For things that are not to be remedied.

*William Shakespeare, King Henry VI, Part I*

Plan to throw one away; you will, anyhow.

*F. P. Brooks, Jr., The Mythical Man-Month*

---



---

## Summary of Problem Solving and Programming Techniques

- There are four commonly used methods for terminating an input loop: ask before iterating, list headed by size, list ended with a sentinel value, and running out of input. (The last one can be dangerous if not handled correctly.)
- Using the boolean `eofn` is one good way to end a loop by “running out of input.”
- One way to design an algorithm is to modify a known algorithm that solves a related problem.
- It is usually best to design loops using pseudocode that does not specify a choice of Pascal looping mechanism. Once the algorithm has been designed, the choice of which Pascal loop statement to use is usually clear.
- The technique of unrolling a loop can be used to redesign loops to get the effect of ending partway through the loop body.
- A *repeat* loop should not be used unless you are certain that the loop should always be iterated at least once.
- A *for* loop can be used to obtain the equivalent of the instruction “repeat the loop body N times.”
- One way to simplify reasoning about nested loops is to make the loop body a procedure.
- Programs should be designed to be robust enough to perform adequately even if the user makes a slight mistake in entering input.
- Always check loops to be sure that the variables used by the loop are properly initialized before the loop begins.
- Never terminate a loop with a test of `real` values for equality. Tests for equality are dangerous with other types as well.
- Always check loops to be certain they are not iterated one too many times or one too few times.
- When debugging loops, always check the first and last iteration of the loop body.
- When you are debugging loops, it helps to trace key variables in the loop body.
- If a program or algorithm is very difficult to understand or performs very poorly, do not try to fix it. Instead, throw it away and start over.

---

## Summary of Pascal Constructs

### while statement

Syntax:

```
while <boolean expression> do
    <body>
```

---

Example:

```
while X < 3 do
  begin
    write(X);
    X := X + 1
  end
```

The <body> can be any Pascal statement. When the *while* statement is executed, the <boolean expression> is first evaluated. If it evaluates to *true*, then the <body> is executed. If the <boolean expression> instead evaluates to *false*, then nothing more happens. This process is repeated again and again. Each time the <boolean expression> evaluates to *true*, the <body> is executed. The first time it evaluates to *false*, the *while* statement terminates and the program goes on to the next item.

### repeat statement

Syntax:

```
repeat
  <statement 1>;
  <statement 2>;
  .
  .
  .
  <statement n>
until <boolean expression>
```

Example:

```
repeat
  Sum := Sum + Next;
  read(Next)
until Next <= 0
```

When the *repeat* statement is executed, the statements are executed in order and then the <boolean expression> is evaluated. If it evaluates to *true*, the *repeat* statement is over, and the program goes on to the next thing. If the <boolean expression> evaluates to *false*, then the statements are repeated again. The list of statements is repeated again and again until the <boolean expression> evaluates to *true*. At that point the *repeat* statement is complete, and the program goes on to the next item.

### coln

Syntax:

```
coln
```

---

Example:

```
repeat
  read (Next) ;
  Sum := Sum + Next
until eoln
```

The end-of-line boolean. Its value becomes `true` after the program reads the last value on a line, using a `read` statement. Executing a `readln` statement resets it to `false` (provided there is some input data on the next line). See the explanation of *end-of-line character* in the chapter text for a more precise description of `eoln`.

### for statement (basic form)

Syntax:

```
for <control variable> := <initial exp> to <final exp> do
  <body>
```

Example:

```
for J := 20 to 500 do
begin
  Sum := Sum + J;
  writeln(Sum)
end
```

<control variable> must be a variable of type `integer`. (Other types are possible and will be discussed in Chapter 8.) Both <initial exp> and <final exp> must be expressions that evaluate to the same type as the <control variable>. The <body> may be any Pascal statement. When the `for` statement is executed, <initial exp> and <final exp> are first evaluated. If the value of <final exp> is less than that of <initial exp>, then the `for` statement terminates without doing anything else. If the value of <initial exp> is less than or equal to the value of <final exp>, the <body> is executed with the value of <control variable> set equal to the value of <initial exp>, and then executed with the value of <control variable> increased by one, and then by one more, and so forth. The last time <body> is executed, the value of <control variable> is equal to the value obtained when <final exp> was evaluated. The <control variable> may not be changed by the <body>; if it is, the effect is unpredictable.

### for statement (downto form)

Syntax:

```
for <control variable> := <initial exp> downto <final exp> do
  <body>
```

Example:

```
for J := 500 downto 20 do
begin
  Sum := Sum + J;
  writeln(Sum)
end
```



Similar to the previous kind of *for* statement, except that in this kind of statement, the <control variable> is decreased by one on each iteration of the loop. If the value of <final exp> is greater than that of <initial exp>, then the *for* statement terminates without doing anything.

---

## Exercises

### Self-Test Exercises

9. What is the output of the following program fragment (when embedded in a complete program)?

```
for I := -3 to 11 do
    write(2 * I)
```

10. What is the output of the following program fragment (when embedded in a complete program)?

```
for I := 10 downto 1 do
    write(I)
```

11. Write a program that outputs all the even numbers between 1 and 25.

12. What is the output of the following program fragment (when embedded in a complete program)?

```
Limit := 3;
for I := 1 to Limit do
    begin
        writeln(I, Limit);
        Limit := 2
    end
```

13. Predict the output of the following nested *for* loop:

```
for J := 10 downto 1 do
    for I := 1 to 10 do
        writeln(I, ' times ', J, ' equals ', I * J)
```

14. For each of the following situations, tell which type of loop (*while*, *repeat*, or *for*) would work best:

- Summing a series with a specific number of terms.
  - Reading in the list of exam scores for one student in a class.
  - Reading in the number of days of sick leave taken by employees in a department. The input consists of a list of numbers giving the number of days missed by each employee who took sick leave.
  - Testing a procedure with different values of the parameters.
-

## Interactive Exercises

15. Replace the *while* loop in Figure 7.2 with the “almost equivalent” *repeat* loop introduced in the section entitled “The Repeat Statement,” and then run the modified program. Since the *repeat* loop is not completely equivalent to the *while* loop, the program will not perform correctly on an empty list. Try running the program on both nonempty and empty lists of numbers. (For the empty list, think of the program as adding the list of sales commissions earned by a lazy salesman for a month in which he only went fishing and never saw a customer.)
16. Embed the nested loop from Exercise 13 in a program and run the program. Try changing the various numbers and running the program again. In each case, predict the output before running the program.
17. Modify any one of the programming exercises you did for Chapter 6 by adding a loop so that the calculation can be repeated again and again with new input as long as the user desires. Do this once with a *repeat* loop. Do it again with a *while* loop, and this time allow the user to change his or her mind and not perform the calculation even once.

## Programming Exercises

18. Write a program to read in a list of test scores and output the highest score, the lowest score, and the average score. Most of the solution can be found in Figures 7.5 and 7.9.
19. Write a program to list the numbers from 0 to 25, their squares, square roots, fourth power, and fourth root. The output should be in a neat five-column format.
20. Write a program that reads in a number  $n$  and then outputs the sum of the squares of the numbers from 1 to  $n$ . If the input is 3, for example, the output should be 14, because

$$1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$$

The program should allow the user to repeat this calculation as often as desired.

21. Write a program to determine the largest value of  $n$  such that your computer can compute  $n!$  without integer overflow.  $n!$  is the product of all positive numbers that are less than or equal to  $n$ . For example,  $3! = 1 \times 2 \times 3 = 6$ . By convention,  $0!$  is set equal to 1.
22. Write a program that takes one real value as input and computes the first integer  $n$  such that  $2^n$  is greater than or equal to the input value. The program should output both  $n$  and  $2^n$ .
23. Write a program that outputs the balance of an account after each succeeding year. The input is the initial balance, the interest rate, the first year, and the last year. Assume that the deposit is made on January first and that the interest for the past year is calculated and paid once each year on January first. Next, modify the program so that it shows three different balances for three different ways of calculating the interest: simple interest, interest compounded semiannually, and interest compounded quarterly.

24. Modify the program in the previous exercise (or do it from scratch) so that interest is compounded monthly (and not any other way, such as semiannually) and so that the output shows the interest at the end of each month. You should insert a `readln` with no arguments to stop the output at the end of each year. When the user hits the return key, the figures for the following year are displayed. If this were not done, the user would not have time to read the output. After that, enhance the program so that the calculation can be made as often as desired with different inputs.

25. Interest on a loan is paid on a declining balance, and hence a loan with an interest rate of, say, 14% can cost significantly less than 14% of the balance. Write a program that takes a loan amount and interest rate as input and then outputs the monthly payments and balance of the loan until the loan is paid off. Assume that the monthly payments are one-twentieth of the original loan amount and that any amount in excess of the interest is credited toward decreasing the balance due. Thus, on a loan of \$20,000, the payments would be \$1000 a month. If the interest rate is 10%, then each month the interest is one-twelfth of 10% of the remaining balance. The first month (10% of \$20,000)/12 or \$166.67 would be paid in interest, and the remaining \$833.33 would decrease the balance to \$19,166.67. The following month the interest would be (10% of \$19,166.67)/12, and so forth. Also have the program output the total interest paid over the life of the loan. Finally, determine what simple annualized percentage of the original loan balance was paid in interest. For example, if \$1,000 was paid in interest on a \$10,000 loan and it took two years to pay off, then the annualized interest is \$500, which is 5% of the \$10,000 loan amount.

26. Write a program to reads in a real number  $x$  and output the integer  $n$  closest to the cube root of  $x$ . Assume that  $x$  is always nonnegative.

27. A *perfect number* is a positive integer that is equal to the sum of all those positive integers (excluding itself) that divide it evenly. The first perfect number is 6, because its divisors (excluding itself) are 1, 2, and 3, and because  $6 = 1 + 2 + 3$ . Write a program to find the first three perfect numbers (include 6 as one of the three).

28. The Fibonacci numbers  $F_n$  are defined as follows:  $F_1$  is 1,  $F_2$  is 1, and  $F_{i+2} = F_i + F_{i+1}$ , for  $i = 1, 2, \dots$ . In other words, each number is the sum of the previous two numbers. The first few Fibonacci numbers are 1, 1, 2, 3, 5, 8. One place that these numbers occur is as certain population growth rates. If a population has no deaths, then the series shows the increase in population after each generation. A generation is the time it takes a member to reach reproducing age. The formula applies most straightforwardly to asexual reproduction at a rate of one offspring per parent per generation. In any event, the green crud population grows at that rate and produces one generation every five days. Hence, if a green crud population starts out as 10 pounds of crud, then in 5 days there is still 10 pounds of crud; in 10 days there is 20 pounds of crud, in 15 days there is 30 pounds, in 20 days there is 50 pounds, and so forth. Write a program that takes as input the initial size of a green crud population (in pounds) and a number of days and then outputs the number of pounds of green crud after that many days. Assume that the population size remains the same for four days and then increases on the fifth day.

---



29. Write a program to find all integer solutions to the equation

$$4x + 3y - 9z = 5$$

for values of  $x$ ,  $y$ , and  $z$  between 0 and 100.

30. Write a program that reads in and evaluates expressions such as

+15+90-100-7+36end

The expression is a sequence of integers, each preceded by a sign. The expression is terminated with the word *end*. There are no blanks in the expression.

31. The value  $e^x$  can be approximated by the sum

$$1 + x + x^2/2! + x^3/3! + \cdots + x^n/n!$$

Write a program that takes a value  $x$  as input and outputs this sum for  $n$  taken to be each of the values 1 to 100. The program should repeat the calculation for new values of  $x$  until the user says he or she is through. Use variables of type *real* to store the factorials, or you are likely to produce integer overflow.

32. Write a procedure that fills one variable parameter of type *integer* with a value read from the keyboard. The procedure will read the input as a string of characters and will check and recover from input typing mistakes as follows: It will skip over all characters other than the ten digits. Thus, it will interpret the input \$12\*\*%5 as the number 125. It will then echo the integer value and allow the user to try again if it is not correct. Assume the number is three or fewer digits long.

---

## References for Further Reading

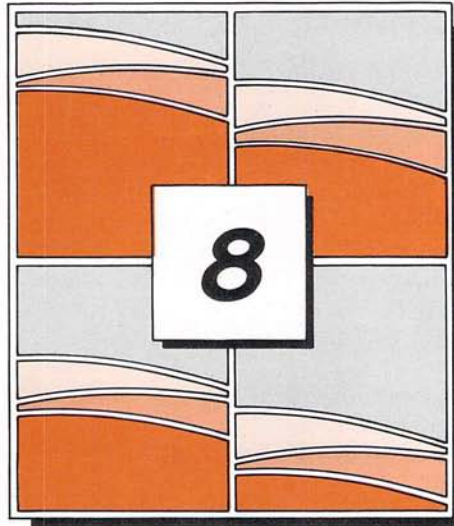
The following books give more detailed discussions of loop invariants (covered in the optional section) and related topics.

S. Alagic and M.A. Arbib, *The Design of Well-Structured and Correct Programs*, 1978, Springer-Verlag, New York.

D. Gries, *The Science of Programming*, 1981, Springer-Verlag, New York.

---





## ***Designing Functions and Data Types***

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.

"Begin at the beginning," the King said, very gravely, "And go on till you come to the end: then stop."

*Lewis Carroll, Alice in Wonderland*



## Chapter Contents

Use of Functions	TURBO Pascal—Uppcase
A Sample Function Declaration	Subrange Types
Case Study—A Function to Compute Powers	TURBO Pascal—The R Compiler Directive
Pitfall—Thinking the Function Name Is a Variable	The For and Case Statements Revisited
Pitfall—Special Cases	Allowable Parameter Types
Local Identifiers	TURBO Pascal—String Parameters
Functions That Change Their Minds	Pitfall—Parameter Type Conflicts
Side Effects	Pitfall—Anonymous Types
Self-Test Exercises	Case Study—Complete Calendar Program
Boolean-Valued Functions	Use of Subrange Types for Error Detection
Case Study—Testing for Primes (Optional)	Enumerated Types (Optional)
More Standard Functions (Optional)	TURBO Pascal—String Functions and Procedures
Random Number Generators (Optional)	TURBO Pascal Case Study—Replacing Substrings
Designing Your Own Pseudorandom Number Generator (Optional)	Summary of Problem Solving and Programming Techniques
A Better Method for Scaling Random Numbers (Optional)	TURBO Pascal—Summary of String Functions and Procedures
TURBO Pascal—A Predefined Random Number Generator (Optional)	Summary of Pascal Constructs
Self-Test Exercises	Exercises
Ordinal Types	References for Further Reading
TURBO Pascal—String Comparisons	
The Functions pred, succ, ord, and chr (Optional)	

**W**e have already encountered a number of predefined standard functions, such as `sqrt`, `trunc`, and `round`. They are automatically provided in the Pascal language. You can also define new functions within a Pascal program. In this chapter we describe how these new functions are designed and used. We then describe how some new kinds of data types can be defined within Pascal. In addition to the predefined types, such as `integer` and `char`, Pascal allows the programmer to define types. In this chapter we introduce some of the simpler kinds of defined data types and show how they can be helpful in detecting program errors. We conclude by describing the predefined functions and procedures which TURBO Pascal provides for manipulating strings.

---

## Use of Functions

Programmer-defined functions are called in exactly the same way as the standard functions we have been using. To *call* any function in Pascal, the program provides the function with one or more arguments. The function then returns a single value. For example, in the statement

```
X := round(2.9)
```

the function `round` returns the value 3. A function call is a particular kind of expression and, like all other expressions, it has a value. That value is said to be the *value returned* by the function.

A Pascal function is used differently from a procedure. A procedure call is a statement and, like any statement, it performs some action. A function call is an expression and, like any expression, returns a value. Although it is called in a different way, you declare a function in much the same way that you declare a procedure. The function declarations and procedure declarations appear in the same place in a program, possibly even intermixed. To get a feel for these function declarations, we will start with a simple example.

---

## A Sample Function Declaration

Figure 8.1 shows a program with a function declaration for a function called `Cube`. This function takes one argument of type `integer` and returns a value of type `integer`; more specifically, it returns the cube of its argument. Function declarations have formal parameter lists that have the same syntax as procedure parameter lists. The sample function has one formal value parameter, which is set equal to the value of the function argument when the function is called. Function arguments are nothing more nor less than parameters. Although in functions, parameters are traditionally called *arguments*, that is simply another word for “parameters.”

Although a function declaration greatly resembles a procedure declaration, it does differ in some ways from a procedure declaration. First of all, it starts with the word *function* rather than the word *procedure*. Two more significant differences are concerned with how the value to be returned by the function is specified. The first significant difference is that the function is given a type in the function declaration heading. In the function heading in Figure 8.1, the second instance of `integer` (the one at the end before the semicolon) tells the compiler that the function `Cube` is of type `integer`. This means that the value returned will be of type `integer` and, hence, that the function can only be used in places where it is appropriate to use a value of type `integer`. The type of the value returned may be any of the types we have seen thus far: `integer`, `real`, `char`, or `boolean`, as well as a few additional types that we will discuss later in this book. The complete syntax for a function heading is given in Figure 8.2.

The second major difference between a procedure declaration and a function declaration is that somewhere within the body of the function declaration, the function

*call*

*value  
returned*

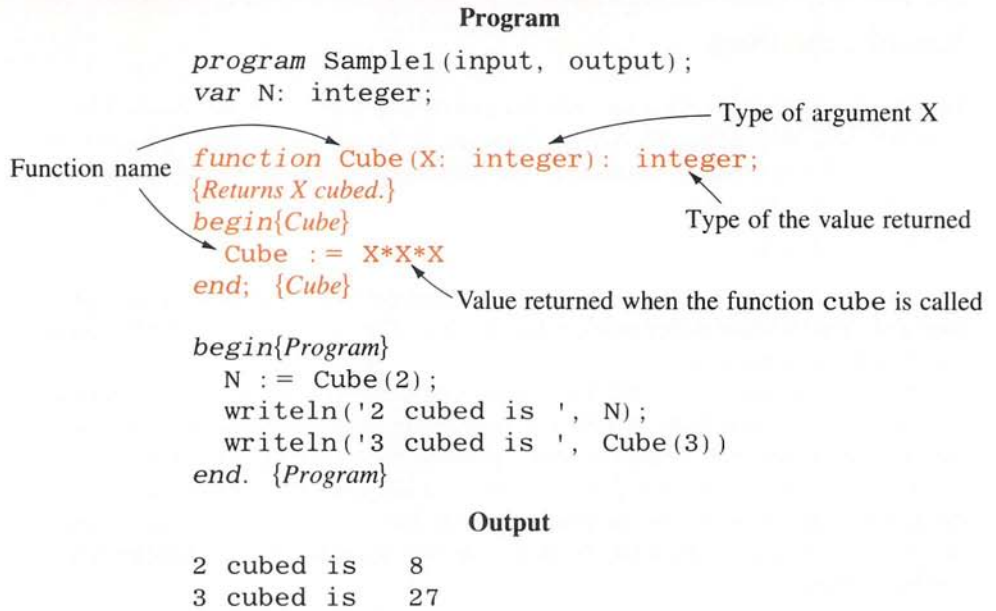
*parameter  
list*

*arguments*

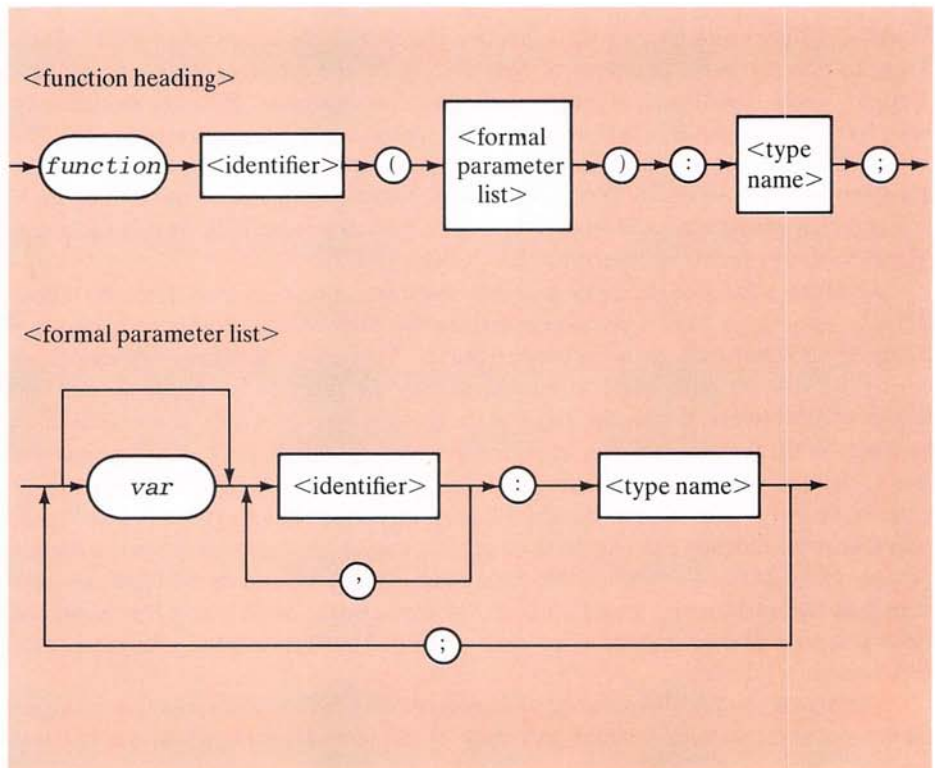
*function  
declaration*

*assignment to  
function name*

---



**Figure 8.1**  
A sample function  
declaration.



**Figure 8.2**  
Syntax for a  
function  
declaration  
heading.



name must appear on the left-hand side of an assignment statement. This is how the function declaration specifies the value to be returned. Although it looks like a simple assignment of a value to a variable, the statement

```
Cube := X*X*X
```

in Figure 8.1 is something very different. The function name, in this example *Cube*, is not a variable, and the assignment statement does not mean what an assignment statement usually means. The only reason it is written in this fashion is that this is the traditional way to write it. The purpose of one of these special assignment statements (with the function name on the left-hand side) is to specify the value that is returned by the function call. In this example, the above assignment statement means to return  $X$  times  $X$  times  $X$  as the value of *Cube*. When the function *Cube* is called in the program statement

```
N := Cube (2)
```

the value of the argument, in this case 2, is used to set the value of  $X$ , and the expression  $X*X*X$  is then evaluated. The statement

```
Cube := X*X*X
```

says to return this value, namely 8, as the value of the function. In the sample program, the value of *Cube* (2), and hence the new value of *N*, is 8. There is no variable named *Cube* that is set to 8 or to any other value. The identifier *Cube* is not the name of a variable. It is the name of a function.

*value  
returned*

---

## Case Study

---

### A Function to Compute Powers

#### Problem Definition

We have a predefined function to compute squares, and we have just written a declaration for a function to compute cubes. Rather than write separate functions for the fourth power, fifth power, and so forth, it makes more sense to have a general-purpose function to compute powers of the form

$$x^n$$

where  $n$  as well as  $x$  is a parameter. Unlike many other programming languages, Pascal has no predefined function of this form, and so we will need to write a function declaration for it. To make the function even more generally applicable, we will allow the argument  $x$  to be of type *real*, but to simplify the problem, we will assume that the argument  $n$  is a nonnegative integer. Hence, if we name the function *Power*, then the function heading must be the following:

```
function Power (X: real; N: integer): real;
{Precondition: N >= 0.
Returns X to the power N; returns 1 when N equals 0.}
```

---

## Discussion

We wish to compute the value

$$X^N$$

One definition of this quantity is the following:

$$\underbrace{X * X * \dots * X}_{N \text{ times}}$$

Many problem definitions are already algorithms, sometimes in a disguised form. To obtain an algorithm for translation into Pascal often requires little more than rephrasing the definition. In this case, the definition gives us an algorithm for computing the value we want: Simply multiply `X` by itself `N` times. To do this we use a *for* loop and a local variable `Product` to hold the partial products:

*algorithm  
from definition*

```
for I := 1 to N do
  begin{for}
    Product := Product*X
    {Product is X to the Power I.}
  end {for}
{Product is X to the Power N.}
```

Notice that the *for* loop is used simply to repeat a statement execution `N` times. The loop control variable `I` is never referenced inside the loop body. This is an example of the “repeat *n* times” construct we discussed in Chapter 7.

*initialize  
value*

Since the variable `Product` appears on the right-hand side of the assignment statement inside the *for* loop, it must be given a value before the loop is executed. The correct value is 1. To see that this is true, notice that if `Product` is initialized with the value 1, then after one loop iteration its value will be `X`, which satisfies the comment assertion inserted before the *end*.

**ALGORITHM**

At this stage the algorithm is complete: Initialize `Product` to 1, and execute the loop. This sets `Product` equal to the desired value. To return this value as the value of the function, we need only add the statement

```
Power := Product
```

The complete function declaration is given in Figure 8.3.

## Pitfall

### Thinking the Function Name Is a Variable

In a function declaration the function name can appear on the left-hand side of an assignment operator, as in the following line from the declaration for the function `Power` shown in Figure 8.3:

(continued, page 274)

**Program**

```

program Test(input, output);
{Tests the function Power.}
var Arg1: real;
    Arg2: integer;
    Ans: char;

function Power(X: real; N: integer): real;
{Precondition: N >= 0.
Returns X to the power N; returns 1 when N equals 0.}
var I: integer;
    Product: real;
begin{Power}
    Product := 1;
    for I := 1 to N do
        begin{for}
            Product := Product*X
            {Product is X to the Power I.}
        end; {for}
    Power := Product
end; {Power}

begin{Program}
    repeat
        writeln('Enter a real and a nonnegative integer. ');
        readln(Arg1, Arg2);
        writeln(Arg1, ' to the power ', Arg2);
        writeln('is ', Power(Arg1, Arg2));
        writeln('Another test? (yes/no) ');
        readln(Ans)
    until (Ans = 'N') or (Ans = 'n')
end. {Program}

```

Type of the value returned

Returns the value of Product as the value of Power (X, N)

**Sample Dialogue**

```

Enter a real and a nonnegative integer.
2.0 3
2.00000000E+00 to the power 3
is 8.000000000000E+00
Another test? (yes/no)
yes
Enter a real and a nonnegative integer.
0.12 2
1.20000000E-1 to the power 2
is 1.440000000000E-2
Another test? (yes/no)
no

```

**Figure 8.3**  
**Function**  
**declaration in a**  
**test program.**



```
Power := Product
```

This leads some programmers to think that the function name can be used as a variable. However, it is not a variable, and it cannot be used as one. For example, the following tempting rewrite for the body of the function declaration in Figure 8.3 is incorrect:

```
begin{Incorrect version of Power}  
  Power := 1;  
  for I := 1 to N do  
    Power := Power * X  
  end; {Incorrect version of Power}
```

The statement `Power := Power * X` will produce an error message. In this situation the compiler is likely to think that the occurrence of `Power` on the right-hand side is some sort of incorrect function call and will issue an error message to this effect. The message is likely to read something like “Insufficient arguments for a function” or “number of parameters does not agree with declaration.”

---

## Pitfall

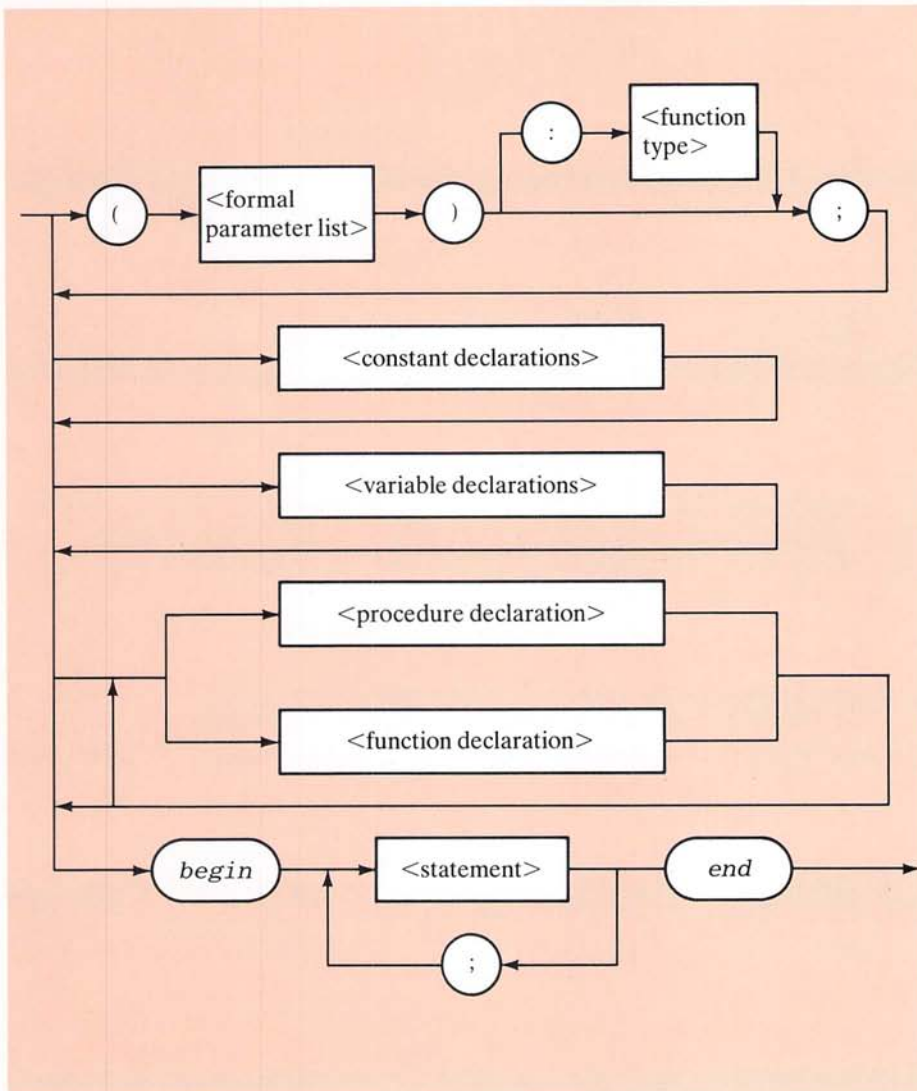
### Special Cases

When designing functions, it is important to pay particular attention to special cases. In the function `Power` in Figure 8.3, any exponent of zero or less will cause the `for` loop to terminate without performing any iterations at all, and so will cause the function to return a value of 1. As the precondition indicates, we are not using this function for negative exponents, and so we need not be concerned with the value returned in such cases. However, we are allowing it to be used with the exponent zero, and so we should check that special case. Since `X` to the power zero is traditionally defined to be one, that is the correct value to return. Special cases such as this should always receive a special check to be sure they are handled correctly.

---

## Local Identifiers

Notice that the function in Figure 8.3 has two local variables. Functions may also have local constants. They may even have local procedures and local functions. These local variables, constants, and so forth are declared and used just as they are in procedures. As with procedures, the local declarations plus the parameter list and the list of state-



**Figure 8.4**  
Syntax for a block.

ments in a function declaration are referred to as a *block*. Figure 8.4 summarizes the order of declarations within a block. This ordering applies to the main block of a program and the block of a procedure declaration, as well as to the block of a function declaration. Observe that the function and procedure declarations may be intermixed in the declaration section of a program or in the local declaration section of a procedure or function. The only restriction is that the declaration for each function or procedure must appear before the function or procedure is called.

**Program**

```

program Test(input, output);
{Tests the function CapYN.}
var Ans: char;

function CapYN(Ans: char): char;
{Precondition: Ans has one of the characters 'y', 'Y', 'n',
or 'N' as its value. Returns the uppercase version of Ans.}
begin{CapYN}
  CapYN := Ans; {tentatively}
  if Ans = 'n' then
    CapYN := 'N'
  else if Ans = 'y' then
    CapYN := 'Y'
  {else the tentative value is used.}
end; {CapYN}

begin{Program}
  writeln('This is a test. ');
  repeat
    writeln('Answer yes or no: ');
    readln(Ans);
    writeln('The uppercase version of ');
    writeln('the first letter you typed is: ', CapYN(Ans));
    writeln('Test again? (y/n) ');
    readln(Ans)
  until CapYN(Ans) = 'N';
  writeln('End of test. ')
end. {Program}

```

**Sample Dialogue**

```

This is a test.
Answer yes or no:
no
The uppercase version of
the first letter you typed is: N
Test again? (y/n)
y
Answer yes or no:
No
The uppercase version of
the first letter you typed is: N
Test again? (y/n)
Yes
Answer yes or no:
yes

```

**Figure 8.5**  
A function that  
changes its mind.



```

The uppercase version of
the first letter you typed is: Y
Test again? (y/n)
y
Answer yes or no:
YES
The uppercase version of
the first letter you typed is: Y
Test again? (y/n)
n
End of test.

```

**Figure 8.5**  
(continued)

---

## Functions That Change Their Minds

Since a function must always return a value, a function declaration must always contain at least one statement that uses the function name on the left-hand side of an assignment operator. Every call of the function must cause at least one of these assignment statements to be executed. A function can contain more than one such statement, and sometimes a function call may cause two such assignment statements to be executed. If two or more assignments to the function name are made when the function is called, then the value returned by the function is the last value assigned to the function name.

For example, consider the function in Figure 8.5. It assumes that its argument is one of the symbols 'y', 'Y', 'n', or 'N'. It returns the uppercase version of this argument; if the argument is 'y' or 'Y', it returns 'Y'; if the argument is 'n' or 'N' it returns 'N'. This function might be used to process yes-or-no answers. For example, to terminate the test loop we used the boolean expression

```
CapYN(Ans) = 'N';
```

instead of

```
(Ans = 'n') or (Ans = 'N')
```

The function declaration for CapYN first makes the following assignment to the function name:

```
CapYN := Ans
```

This is a tentative choice for the value returned. If the value of Ans were 'N', this would cause the value of Ans, namely 'N' to be returned. If however, the value of the argument Ans were instead the lowercase letter 'n', then the value returned would be recomputed by the following second assignment to the function name:

```
if Ans = 'n' then
  CapYN := 'N'
```

*example*

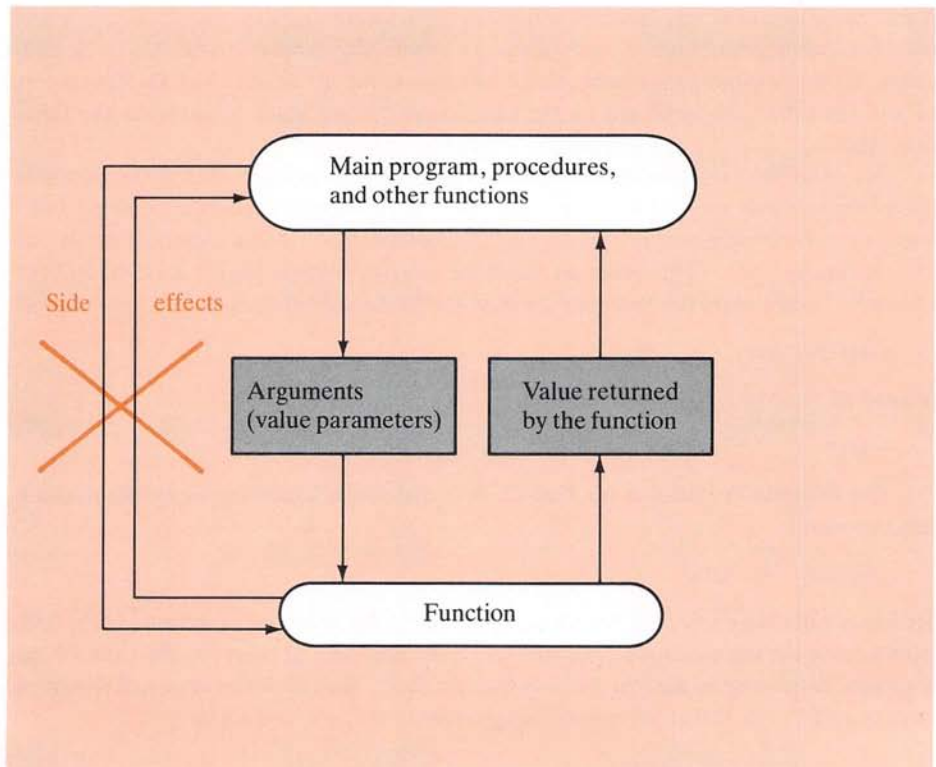
## Side Effects

*what a  
function  
can do*

The purpose of a function is to take the values of some arguments and return a *single* value, but a Pascal function can actually do much more. In Pascal a function is a procedure with the added feature of returning a value. Hence, a function can do anything that a procedure can do. All our examples used value parameters as arguments. This is typical, but a function is allowed to have variable parameters as well as value parameters as its arguments. A function can change the value of a global variable, write a message to the screen, read a value from the keyboard, or do anything else that a procedure can do, so long as it also returns a value. However, it is usually a bad idea to use any of these other features when designing functions.

*what a  
function  
should not do*

If a function changes the value of a global variable, sets the value of a variable parameter, or causes a `read` or `write` statement to be executed, that extra feature is referred to as a *side effect*. As indicated in Figure 8.6, side effects are usually a bad idea. We think of a function as simply returning a value, and if we want it to do more, it is usually clearer to write a procedure rather than a function and to return the value via a variable parameter.



**Figure 8.6**  
Side effects of a  
function.

---

With all appliances and means to boot.

*William Shakespeare, King Henry IV, Part III*

---

## Self-Test Exercises

1. What is the output of the following program?

```
program Exercisel (input, output);

function Crazy (N: integer): char;
begin{Crazy}
    Crazy := 'X';
    if N = 1 then
        Crazy := 'A';
    if N = 2 then
        Crazy := 'B'
end; {Crazy}

begin{Program}
    writeln (Crazy (1), Crazy (2), Crazy (3))
end. {Program}
```

2. What is wrong with the following function declaration?

```
function TwoPower (N: integer): integer;
{Returns 2 to the power N. Precondition N >= 0.}
var I: integer;
begin{TwoPower}
    TwoPower := 1;
    for I := 1 to N do
        TwoPower := TwoPower * 2
end; {TwoPower}
```

3. Write a function that has two arguments, one for the length and one for the width of a rectangle, and that returns the area of a rectangle with those dimensions.
4. Write a function declaration for a function that has one argument of type integer and that returns the letter 'P' if the number is positive and the letter 'N' if it is zero or negative.
5. For each of the following subtasks, tell whether it is best implemented as a function or as a procedure.
- Converting a given number of miles to an equivalent number of kilometers.
  - Converting a given number of centimeters to feet and inches.



- c. Displaying the output of a program on the screen.
- d. Computing the net income and the tax due on an income tax return, given the gross income and the adjustments to income.
- e. Computing an automobile's mileage per gallon, given the miles traveled and the number of gallons of gasoline consumed.

---

## Boolean-Valued Functions

A function may return a value of type `boolean`. A call to the function can then be used anywhere that a boolean expression is allowed. This can often help to make a program easier to read. By means of a function declaration, we can associate a complex boolean expression with a meaningful name and use that name as the boolean expression in an *if-then*, *if-then-else*, *repeat*, or *while* statement. The result can often make a program read rather like English. For example, the statement

```
if ((Rate >= 10) and (Rate < 20)) or (Rate = 0) then
    begin{then}
        . . .
    end {then}
```

can be made to read as follows:

```
if Appropriate(Rate) then
    begin{then}
        . . .
    end{then}
```

provided the following boolean function has already been defined:

```
function Appropriate(Rate: integer): boolean;
begin{Appropriate}
    Appropriate :=
        ((Rate >= 10) and (Rate < 20)) or (Rate = 0)
end; {Appropriate}
```

---

## Case Study

---

### Testing for Primes (Optional)

#### Problem Definition

In this section we will design a function that has one integer argument, which it tests to see whether the integer is a prime number or not. The function will return `true` if it is prime and `false` if it is not. A *prime number* is an integer that is greater than one and

---

that has no divisors other than itself and one. In other words, the prime numbers are the positive integers (other than one) that cannot be factored. For example, 12 can be factored into 2 times 6, and so is not prime. The first few primes are 2, 3, 5, 7, 11, and 13. As is common when discussing prime numbers, we will confine our attention to positive integers.

## Discussion

We must first design an algorithm to test a positive integer for primality. A candidate number  $N$  is prime provided it has no divisors other than itself and one. Obviously, we cannot check all the positive integers to see if any of them divide  $N$ . Fortunately for us, we know that any divisor of  $N$  must be less than or equal to  $N$ , and so we need only test those integers  $I$  which are less than  $N$ . This observation yields the heart of our algorithm:

1. For each  $I$  such that  $1 < I < N$ , test whether  $I$  is a divisor of  $N$ .
2. If *any such*  $I$  is a divisor of  $N$ , then report that  $N$  is *not* a prime. If *no such*  $I$  divides  $N$ , then report that  $N$  is a prime.

*ALGORITHM  
first version*

This function, like many functions, has some arguments that are special cases. For example, the definition of prime numbers makes a special case of one. It has no divisor other than one and itself. Yet by definition it is not a prime. (The reason has to do with the theory of numbers and is not a topic for this book.) Although one is not a prime, it technically satisfies the basic test that will become the heart of our algorithm. Substituting 1 for  $N$ , we see that there are no numbers  $I$  such that

*special  
cases*

$$1 < I < 1$$

and such that  $I$  divides 1; in fact, there are no values for  $I$  that even satisfy the inequality, and so there are certainly none that both satisfy the inequality and divide 1. Hence, we must add a special case for the argument 1; otherwise, our function will incorrectly claim that it is a prime. Our algorithm will include a special check to see if its argument is 1, and if it is, it will return the value `false`.

This leaves us with the numbers 2 and greater. One should always test the extreme values used by an algorithm to see if they are special cases. The argument 2 is a special case. If we apply the number 2 to the heart of our algorithm, it says to test all integers  $I$  such that

$$1 < I < 2$$

Since there is no such  $I$ , the test is suspect. The number 2 is a prime, and, if applied rigorously, the heart of our algorithm will say that it is a prime. However, to make our algorithm easier to understand, we will make a special case of 2. The arguments 3 and larger behave normally with no subtleties when we apply the heart of our algorithm to them. Our algorithm now takes shape in more detail:

1. If  $N$  is 1, then  $N$  is not a prime.
2. If  $N$  is 2, then  $N$  is a prime.
3. If  $N > 2$  then do the following:
  - 3-1. For each  $I$  such that  $1 < I < N$ , test whether  $I$  is a divisor of  $N$ .

*ALGORITHM  
second version*

- 3-2. If *any such*  $I$  is a divisor of  $N$ , then report that  $N$  is not a prime. If *no such*  $I$  divides  $N$ , then report that  $N$  is a prime.

*refining  
the algorithm*

The following boolean expression is true if  $I$  is a divisor of  $N$ , and is false otherwise:

$$N \bmod I = 0$$

So step 3 can be rewritten as follows:

```
tentatively assume N is a prime;
for I := 2 to N - 1 do
  if (N mod I = 0) then
    change your mind and decide that N is not prime.
```

*guilty  
until  
proven  
innocent*

This is an example of the *guilty-until-proven-innocent* technique. We need to test a list to see if any member of the list is a witness to prove that  $N$  is not prime (“not guilty”). If there is no witness to prove that  $N$  is not prime, then it is assumed (correctly) to be prime. The guilty-until-proven-innocent technique is naturally implemented using a boolean variable, in this case, one called `GuiltyOfPrimality`.

**ALGORITHM**  
*third version*

```
GuiltyOfPrimality := true;
for I := 2 to N - 1 do
  if (N mod I = 0) then
    GuiltyOfPrimality := false
```

*changing  
the value  
returned*

In this case, the answer is the value returned by a boolean-valued function, and since Pascal functions can change their mind about the value (“verdict”) returned, we can simply use the function name rather than a boolean variable. If we name our function `Prime`, then the final Pascal code for this test is

*function  
code*

```
Prime := true;
for I := 2 to N - 1 do
  if (N mod I = 0) then
    Prime := false
```

The value returned is tentatively set equal to `true`. If a divisor is not found, then the value `true` is returned. If a divisor is found, then the function changes its mind and returns `false`. The complete function declaration, embedded in a test program, is displayed in Figure 8.7. The function declaration is now correct but could be made more efficient. Exercise 23, at the end of this chapter, suggests a way of improving the efficiency of the function.

### Program

```
program TestPrimes(input, output);
var N: integer;

function Prime(N: integer): boolean;
{Returns true if N is a prime; otherwise returns false. Precondition: N > 0.}
var I: integer;
```

**Figure 8.7**  
**(Optional)**

**A boolean function  
that tests for prime  
numbers.**



```
begin{Prime}
  if N = 1 then
    Prime := false
  else if N = 2 then
    Prime := true
  else
    begin{N > 2}
      Prime := true; {tentatively}
      for I := 2 to N - 1 do
        if (N mod I = 0) then
          Prime := false
      end {N > 2}
    end; {Prime}

begin{Program}
  writeln('Enter a positive integer and');
  writeln('I will tell you if it is prime. ');
  writeln('Enter a zero to quit. ');

  writeln('Enter an integer: ');
  readln(N);
  while N > 0 do
    begin{while}
      if Prime(N) then
        writeln(N, ' is a prime. ')
      else
        writeln(N, ' is not a prime. ');
        writeln('Enter an integer: ');
        readln(N)
      end; {while}

    writeln('End of program. ')
  end. {Program}
```

#### Sample Dialogue

```
Enter a positive integer and
I will tell you if it is prime.
Enter a zero to quit.
Enter an integer:
15
    15 is not a prime.
Enter an integer:
17
    17 is a prime.
Enter an integer:
0
End of program.
```

Figure 8.7  
(continued)

Name	Description	Type of argument	Type of result	Example	Value of example
arctan	arctangent	real or integer	real	arctan(1.0)	0.785 (radians)
cos	cosine	real or integer	real	cos(0.78) (argument in radians)	0.71091
sin	sine	real or integer	real	sin(1.57) (argument in radians)	1.00
exp	exponential	real or integer	real	exp(2)	$e^2$ (not Pascal notation)
ln	natural logarithm	real or integer	real	ln(2.71828) ( $e$ is approx. 2.71828)	1.000
odd	odd/even function	integer	boolean	odd(5) odd(4)	true false

**Figure 8.8**  
**(Optional)**  
Some more  
predefined Pascal  
functions.

## More Standard Functions (Optional)

Figure 8.8 is a list of some more standard functions. If you have occasion to use trigonometric functions or logarithmic functions, you should use these instead of defining your own versions. Be sure to note that the standard trigonometric functions must have their arguments expressed in radians rather than in degrees. There are  $2\pi$  radians in a complete circle of 360 degrees. The exponential function `exp` and the logarithm function `ln` use the number  $e$  as their base. Thus, `exp(3)` has  $e$  cubed as its value. The number  $e$  is approximately equal to 2.71828.

## Random Number Generators (Optional)

Suppose you flip a coin, write down zero if it comes up heads, and write down one if it comes up tails. You have just made a *random* choice between zero and one. If you roll a

single die and count the number of dots on the top face, you will get a random number between one and six. These are two ways of generating random numbers. There are numerous occasions when a computer program needs, or at least can profitably use, a source of random numbers. Perhaps the most obvious example is that of a game-playing program. Random numbers are also used in simulation programs. A program to model the performance of a proposed new highway interchange would typically use a random number generator to model the arrival times of vehicles. A program to write poetry might use a random number generator to guide the choice of words. These are just a few of the numerous uses for random number generators.

An exact definition of what constitutes a *true random number generator* is a matter of significant philosophical debate. However, the examples of the coin flip and the die provide an adequate feel for the concept. Computer programs typically do not use true random number generators. Instead, they use procedures or functions that generate sequences of numbers that appear to be random. Since these sequences are generated by procedures or functions, they are not “truly” random. Hence, these generators are referred to as *pseudorandom number generators*.

*pseudorandom  
numbers*

For most applications, these pseudorandom number generators are a close enough approximation to a true random number generator. In fact, they are usually preferable to a true random number generator. A pseudorandom number generator has one important advantage over a true random number generator: The sequence of numbers it produces is repeatable. If run twice with the same initial conditions, a pseudorandom number generator will always produce exactly the same sequence of numbers. This can be very handy for a number of purposes. It is very useful for debugging. When an error is discovered, the proposed program changes can be tested with the *same* sequence of pseudorandom numbers that exposed the error. Similarly, a particularly interesting run of a simulation program may be reproduced, provided a pseudorandom number generator was used. With a true random number generator, every run of the program is likely to be different.

In the next section, we describe the most common methods for designing and using pseudorandom number generators. These methods will work reasonably well on any Pascal system. However, some systems have been extended to include a predefined pseudorandom number generator. These predefined generators can be carefully tuned to run efficiently and to produce the most “random looking” sequences possible, given the limitations of the particular computer system. If your system includes such a predefined generator, it makes sense to use it instead of defining your own generator. Later in this chapter, we include a section that describes such a predefined pseudorandom number generator for TURBO Pascal. If you are using TURBO Pascal, it makes sense to use this generator, but you may still want to read the following, optional section to see how such a generator might be defined.

One disadvantage of the predefined TURBO Pascal pseudorandom number generator is that it does not allow as much control over the sequence of random numbers as is sometimes wanted. In particular, it is not possible to repeat an interesting sequence of random numbers (unless the program stores the entire sequence in variables or in a file as the sequence is generated). For many applications this repeatability is not needed, but if you do wish to have this repeatability feature, you will need to design your own generator in the manner discussed in the optional section which follows.



## Designing Your Own Pseudorandom Number Generator

(Optional)

*linear  
congruence  
method*  
  
*seed*

The most common method of generating pseudorandom numbers is the *linear congruence method*. This method starts out with a number called the *seed*. For each individual run of the program, the seed is usually chosen by the user. It completely determines the sequence of numbers produced. There are three other numbers called the Multiplier, the Increment, and the Modulus, which are fixed constants. The formula for generating what are hopefully random-looking numbers is quite simple:

The first number is

$$(\text{Multiplier} * (\text{the seed}) + \text{Increment}) \bmod \text{Modulus}$$

The  $n + 1$ st number is

$$(\text{Multiplier} * (\text{the } n\text{th number}) + \text{Increment}) \bmod \text{Modulus}$$

For example, suppose we take the Multiplier to be 2, the Increment to be 3, and the Modulus to be 5. With a seed value of 1, this produces the following sequence of numbers:

$$\begin{aligned} (2*1 + 3) \bmod 5 &= 0, & (2*0 + 3) \bmod 5 &= 3, \\ (2*3 + 3) \bmod 5 &= 4, & (2*4 + 3) \bmod 5 &= 1, \\ (2*1 + 3) \bmod 5 &= 0, & \dots \end{aligned}$$

The pattern of numbers produced is thus 0, 3, 4, 1, 0, 3, 4, 1, 0, 3, 4, 1, . . .

Something is not right. This formula should produce a sequence that looks like a sequence of randomly chosen integers between zero and one less than the modulus. In this example, the numbers should range from 0 to 4. But the above pattern is a repeating one and does not contain 2. Hence, the value 2 will never be produced. Changing the seed will not help much. If we use 2 as the seed, we obtain the sequence 2, 2, 2, 2, . . . This certainly produces the value 2, but generates no other numbers. The problem is in the choice of the other constants and not in the choice of the seed.

Any choice of values for the constants in our pseudorandom number generator will produce a sequence that ultimately falls into a repeating pattern. However, if the constants are chosen carefully, the pattern will be large and will appear to be random. The values given in Figure 8.9 should work reasonably well on any implementation with a `maxint` value of 32761 or larger. With the constants used in the figure, the random number generator will produce 729 numbers before repeating a number.

Notice that the function `Random` has a variable parameter called `Memory`, which it uses to remember one number. The initial value of this variable is the seed. Each time the function is called, this variable parameter is changed, which violates our guideline that functions should not have side effects. This is one of those rare occasions when it is acceptable to violate the guideline. In this case, a variable parameter, or

```

function Random(var Memory: integer): integer;
{Returns a pseudorandom number between 0 and (Modulus-1);
Memory is changed to a value in this range with each call of the function.
Precondition: The value of Memory is between 0 and (Modulus-1).}
const Modulus = 729;
      Multiplier = 40;
      Increment = 3641;
begin{Random}
  Memory :=
    (Multiplier*Memory + Increment) mod Modulus;
  Random := Memory
end; {Random}

```

**Figure 8.9**  
(Optional)  
A pseudorandom  
number generator.

something like it, is necessary. The generator must somehow remember the last value it produced, since that is what determines the next value it will produce.

Very few programs require an integer chosen at random from the range 0 through 728 (the value of `Modulus-1`). Usually it is a different and smaller range. For example, a program might need a pseudorandom integer between 0 and 10. One possibility is to use the following formula:

$$\text{Random}(\text{Memory}) \bmod 11$$

The procedure `ZeroToTen` in Figure 8.10 uses this formula. The program shown there uses the pseudorandom numbers to produce random-looking output.

Random numbers can be scaled by additive constants. One way to get a pseudorandom number in the range 1 to 10 is to add one to an expression that yields numbers in the range 0 to 9. The following is one such expression:

$$(\text{Random}(\text{Memory}) \bmod 10) + 1$$

The function `Random` shown in Figure 8.9 will produce a pseudorandom number in the range 0 to 728. But what if you want a real value in the range zero to one? Simply divide by the largest value that our random number generator produces. The following function uses this technique to return a pseudorandom value between zero and one. To get a real value in any other range, multiply by an appropriate factor and, if need be, add an appropriate constant.

```

function RandomReal(var Memory: integer): real;
{Returns a pseudorandom real value between zero and one.
Calls the function Random given in Figure 8.9.
Precondition: The value of Memory is between 0 and Max (see const declaration).
The function changes the value of Memory to another value in this range.}
const Max = 728; {The largest value returned by Random.}
begin{RandomReal}
  RandomReal := Random(Memory) / Max
end; {RandomReal}

```

changing  
the range

pseudorandom  
reals

**Program**

```
program RatingGame(input, output);  
const Width = 2; {field width for numbers 1 to 10.}  
var Memory, Rating: integer;  
    Ans: char;  
  
function Random(var Memory: integer): integer;  
{Returns a pseudorandom number between 0 and (Modulus-1);  
Memory is changed to a value in this range with each call of the function.  
Precondition: The value of Memory is between 0 and (Modulus-1).}
```

<The rest of the declaration is given in Figure 8.9.>

```
function ZeroToTen(var Memory: integer): integer;  
{Returns a pseudorandom number between 0 and 10 (0 and 10 are possible).}  
begin{ZeroToTen}  
    ZeroToTen := Random(Memory) mod 11  
end; {ZeroToTen}  
  
begin{Program}  
    writeln('Hi, my name is Gollum. ');  
    writeln('I'm a perfect 10. What are you? ');  
    writeln('Answer with a number between 0 and 10. ');  
    readln(Memory);  
    if Memory > 7 then  
        writeln('Not bad. ');  
    else  
        writeln('Oh well. ');  
  
    writeln('Would you like to rate some other people? (yes/no) ');  
    readln(Ans);  
    while (Ans = 'y') or (Ans = 'Y') do  
        begin{while}  
            writeln('You name somebody and I'll give a rating. ');  
            readln;  
            Rating := ZeroToTen(Memory);  
            writeln('That individual is a ', Rating : Width);  
            writeln('Want to rate somebody else? (yes/no) ');  
            readln(Ans)  
        end; {while}
```

**Figure 8.10**  
(Optional)

Program using a  
pseudorandom  
number generator.

---



```
writeln('Before I leave, let me rate you. ');
Rating := ZeroToTen(Memory);
writeln('I'd say you were a ', Rating :Width)
end. {Program}
```

### Sample Dialogue

Hi, my name is Gollum.  
 I'm a perfect 10. What are you?  
 Answer with a number between 0 and 10.  
**10**  
 Not bad.  
 Would you like to rate some other people? (yes/no)  
**yes**  
 You name somebody and I'll give a rating.  
**Joseph Cool**  
 That individual is a 0  
 Want to rate somebody else? (yes/no)  
**yes**  
 You name somebody and I'll give a rating.  
**Olivia Safran**  
 That individual is a 10  
 Want to rate somebody else? (yes/no)  
**yes**  
 You name somebody and I'll give a rating.  
**Walter Savitch**  
 That individual is a 4  
 Want to rate somebody else? (yes/no)  
**no**  
 Before I leave, let me rate you.  
 I'd say you were a 4

**Figure 8.10**  
(continued)

---

## A Better Method for Scaling Random Numbers (Optional)

Producing pseudorandom numbers in a specified range requires some scaling of the values produced by the function `Random`. The methods of scaling that we have already seen will work well in most situations. However, they do occasionally produce unsatisfactory sequences, and so we will discuss another slightly more complicated, but preferable, method of scaling that avoids such problems. But before discussing the solution, we will illustrate the problem.

Sequences of pseudorandom numbers produced by the linear congruence method have a pattern that can become apparent when the numbers are scaled in certain inno-

---

cent-looking ways. For example, one way to obtain a pseudorandom number chosen from among the three values 0, 1, and 2 is to use the following formula:

`Random (Memory) mod 3`

If this formula is used with `Memory` initialized to the seed value of 100, the resulting sequence will be: 0, 2, 1, 0, 2, 1, 0, 2, 1, . . . The sequence repeats after just three numbers.

Patterns like the one we just saw frequently depend on the last digits of the numbers. One way to break the pattern, therefore, is to discard the last digit. The following formula yields pseudorandom numbers produced by the generator `Random` but with the last digit discarded:

`Random (Memory) div 10`

For example, if `Memory` has a value of 100, then `Random` produces a value of 114, which yields a final value of 11 after applying the `div` operator.

If we combine the two tricks of discarding the last digit and then scaling by applying a `mod 3`, we get the following formula for producing pseudorandom numbers in the range 0 to 2:

`(Random (Memory) div 10) mod 3`

Again starting with a seed value of 100, this more complicated formula produces a more random-looking sequence that starts out 2, 0, 1, 2, 0, 2, 2, 2, 0, 1, . . .

## TURBO Pascal

### A Predefined Random Number Generator (Optional)

TURBO Pascal includes a predefined pseudorandom number generator. It is named `random`, and it is similar to, but not the same as, the function we defined in Figure 8.9. In TURBO Pascal you can choose to use the predefined version of `random`, in which case you need give no function declaration, or you can choose to redefine the function `random` by giving a declaration such as the one in Figure 8.9. In this section we will describe the predefined version of `random`.

*random  
integers*

The predefined function `random` has one argument of type `integer` and returns a pseudorandom `integer` value that is less than this argument. The call

`random (<number>)`

returns a pseudorandom `integer` value greater than or equal to 0 and less than `<number>`. `<number>` is a value parameter of type `integer` and should have a positive value. For example, the following will output two pseudorandom numbers between 0 and 9, possibly including 0 and/or 9:

```
writeln(random(10));
writeln(random(10))
```

You may obtain pseudorandom integers in any ranges by choosing an appropriate argument for `random` and adding an appropriate constant. For example, the following expression returns a pseudorandom number that is greater than or equal to 2 and strictly less than 7:

```
random(5) + 2
```

To output a value of type `real`, the function `random` is called without any argument. So,

```
writeln(random);
writeln(random)
```

*random  
reals*

will output two pseudorandom values of type `real`, each between zero and one. The `real` value returned is always between zero and one.

Figure 8.11 is a TURBO Pascal program that illustrates the use of the predefined pseudorandom number generator `random`.

---

Chance is a word void of sense;  
nothing can exist without a cause.  
*Voltaire, A Philosophical Dictionary*

---

## Self-Test Exercises

6. What is the output produced by the following program?

```
program Exercise6(input, output);
var George: boolean;

function Blaise(N: integer): boolean;
begin{Blaise}
  Blaise := N < 0
end; {Blaise}

begin{Program}
  if Blaise(5) then
    writeln('Hello')
  else
    writeln('Hi');
  George := .Blaise(-8);
  if George then
    writeln('Good-Bye')
  else
    writeln('So long')
end. {Program}
```



**Program**

```

program RatingGame;
{Works in TURBO Pascal but not in standard Pascal.}
var Rating: integer;
    Ans: char;
begin{Program}
    repeat
        writeln('You name somebody and I'll give a rating. ');
        readln;
        Rating := random(11);
        writeln('That individual is a ', Rating);
        writeln('Want to rate somebody else? (Yes or No) ');
        readln(Ans)
    until (Ans = 'N') or (Ans = 'n');

    writeln('Before I leave, let me rate you. ');
    Rating := random(11);
    writeln('I'd say that you were a ', Rating)
end. {Program}

```

**Sample Dialogue**

You name somebody and I'll give a rating.  
**Prostetnic Vagon Jeltz**  
 That individual is a 2  
 Want to rate somebody else? (Yes or No)  
**yes**  
 You name somebody and I'll give a rating.  
**the inventor of the Infinity Improbability Drive**  
 That individual is a 10  
 Want to rate somebody else? (Yes or No)  
**No**  
 Before I leave, let me rate you.  
 I'd say that you were a 7

**Figure 8.11**  
**TURBO Pascal**  
**pseudorandom**  
**number generator.**  
**(Optional)**

7. Write a boolean function of two integer arguments that returns true if the first argument evenly divides the second and returns false otherwise.
8. Write a boolean function declaration for a function InOrder that has three integer arguments and returns true if the three integer arguments are in ascending order. For example, InOrder(1, 2, 3) should return true, and InOrder(1, 3, 2) should return false.
9. (This exercise uses the optional sections "Random Number Generators" and "Designing Your Own Pseudorandom Number Generator.") Write a function that returns a

pseudorandom number chosen from the even numbers between 2 and 20. Use the random number generator developed in this chapter.

## Ordinal Types

The most straightforward way to specify the values of a data type is to list them. In the case of the type `boolean`, this is easy to do. There are just two values: `false` and `true`. In the case of the type `char`, the list is much longer: `...`, `'A'`, `'B'`, `'C'`, `...`. In the case of the type `integer`, the list is so long that it would be unrealistic to write it out, but in principle the values could all be listed. In Pascal, a type whose values are specified by a list is called an *ordinal type*. The types `integer`, `boolean`, and `char` are all ordinal types.

Since the items of any list are ordered, the values of an ordinal type have an order determined by their order in the list. The operation `<` can be used to test this ordering. It is an operation defined for all Pascal ordinal types, and it always yields a `boolean` value. The most obvious case of this is the ordinal type `integer`. In that case, the order is the usual less than ordering of the integers. For example, `1 < 2` and `-5 < -3` both evaluate to `true`. In the case of the type `boolean`, `false` is considered to be less than `true`. There is little intuitive meaning to this ordering. It is completely arbitrary. Nonetheless, it is the prescribed ordering, and so `false < true` evaluates to `true`.

The ordering of the type `char` is more or less the obvious one. Letters are ordered alphabetically, and so `'A' < 'B'` and `'A' < 'Z'` both evaluate to `true`, whereas `'Z' < 'X'` evaluates to `false`. The digits are also ordered as you would expect. For example, `'1' < '2'` and `'0' < '9'` both evaluate to `true`, whereas `'3' < '2'` evaluates to `false`. The ordering of the digits reflects the ordering of the numbers they stand for. What about the character pairs that do not have a traditional ordering? Is the semicolon less than the comma or greater than the comma? Is uppercase `'A'` less than or greater than lowercase `'a'`? Pascal does not say. It does specify that the uppercase letters are ordered alphabetically among themselves, the lowercase letters are ordered alphabetically among themselves, and the digits are ordered in the obvious way. Moreover, it says that the digits are contiguous; that is, there is no character between two intuitively adjacent digits such as `'4'` and `'5'`. Within these constraints, the compiler can be implemented in any way the designer finds convenient. So on one system you may find that `'a' < 'A'` evaluates to `true`, whereas on another system it evaluates to `false`. The moral is clear: Avoid using any ordering properties that can change from system to system.

The type `char` is a good example of a type that is only partially determined by the definition of the language. This is quite common. The usual thing to do is to require a data type to have the properties that programmers normally expect and need but to let the compiler writer do whatever is convenient with the unspecified properties. Although it may be incomplete, the definition of a data type should not be vague. Pascal specifies that the order of the digits is exactly `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`. Similarly, the ordering of any two uppercase letters is rigidly specified, and so forth. Whatever is specified is specified precisely.

*integer*

*boolean*

*char*

The ordering properties of ordinal types can be used in a program. For example, it is sometimes useful to test two letters to determine whether they are in alphabetic order. An obvious example of this would be in a program that alphabetizes a list of words. We have already seen that there are numerous reasons why a program might compare two integer values.

When comparing two values of an ordinal type, the variations on the  $<$  operation, namely  $<=$ ,  $>$ , and  $>=$ , are all available and have the obvious interpretation.

*real*

Of all the standard Pascal types we have seen thus far, only the type `real` is not an ordinal type. This is a consequence of the abstract model of the real numbers that is used in mathematics. In that model, the real numbers cannot be listed. Computers can represent only a finite number of `real` constants, and so the `real` values for any particular implementation can, in principle, be listed. However, the definition of Pascal does not specify the list; moreover, we do not normally think of the `real` values as being on a list. For these reasons the type `real` is not considered to be an ordinal type. As we have seen, however, the comparison operators, such as  $<$ , can be applied to values of type `real`, in the same fashion in which they can be applied to values of ordinal types.

*simple  
types*

In many, but not all, ways the type `real` is like an ordinal type. The unifying term *simple type* is used to refer to any type that is either the type `real` or is an ordinal type. The type `real` is both important and troublesome. All of Chapter 15 is devoted to a discussion of this type.

## TURBO Pascal

### String Comparisons

The *string* types in TURBO Pascal, such as `string[20]`, are not ordinal types, although string values are ordered and may be compared using  $<$  and the other ordering relations. The order is roughly alphabetic. In a comparison of two strings, which are either both all uppercase letters or both all lowercase letters, the order is alphabetic. For other cases the rule is more complicated.

The comparison operator with arbitrary string values is not used very often, but we include the complete rule for those who want or need it:

#### Comparison Rule for Strings

The two strings are compared from left to right until two different characters are encountered. The order is then determined by the order of these two symbols. If the two strings are the same except that one is shorter than the other, then the shorter one is "less than" the longer one.

The order of the individual symbols is given in the ASCII character table in Appendix 21. In this order all uppercase letters are "less than" all lowercase letters, and the blank is "less than" all of the letters. As examples, all of the following evaluate to true:



```
'baby' < 'booby' {because 'a' < 'o'}
'fool' < 'foolish'
{because 'foolish' starts with 'fool' but has more symbols.}
'WISE' < 'wise' {because 'W' < 'w'}
'Wise' < 'wise' {because 'W' < 'w'}
'ACTOR' < 'ADDER' {because 'C' < 'D'}
'The gang' < 'Their gang' {because(the blank) < 'i'}
```

---

## The Functions `pred`, `succ`, `ord`, and `chr` (Optional)

Since the values of an ordinal type are ordered in a list, a programmer may find it useful to refer to the value preceding a given value, or to the value following a given value in this list. Pascal provides two functions, `pred` and `succ`, for exactly these purposes. Given a value of some ordinal type as an argument, such as 18, the function `pred` returns the preceding value of that type and the function `succ` returns the succeeding value in the ordering of that type. For example, `pred(18)` returns 17, and `succ(18)` returns 19. Similarly, `succ('B')` usually returns 'C', and `pred('B')` usually returns 'A'. Since the exact ordering of the characters may vary from system to system, the values returned by `pred` and `succ` may vary somewhat from system to system, but we will assume that these are the values returned.

*pred*  
*succ*

Since the values of an ordinal type are listed in a certain order, we can meaningfully ask where a specific value is located on the list. The Pascal function `ord` provides a way of doing this. However, the ordinal types are usually numbered starting with zero rather than one. So `ord` applied to the first value of an ordinal type returns 0; when applied to the second value, it returns 1, and so forth. The type `integer` is the sole exception to this numbering scheme. The function `ord` applied to any value of type `integer` simply returns that value. For example, `ord(-5)` returns -5, which makes `ord` an uninteresting function to apply to integers.

*ord*

Since it is the type `char` that is ultimately used when a program communicates with the outside world, it is a special type in a number of ways. One of its special features is a standard function that is the inverse of `ord`. Given a nonnegative integer value, the function `chr` returns the character value in that position on the list of values of type `char`. So `chr(0)` returns the first value of type `char`, `chr(1)` returns the next value, `chr(2)` returns the next value after that, and so forth. For numbers that do not correspond to any character value, `chr` is undefined.

*chr*

These four functions, which depend on the ordering of ordinal types, are occasionally useful but are neither essential nor even very widely used. Only a few specialized uses are very common. One use involves text processing. Although it is not part of the definition of Pascal, most systems order the letters so that the uppercase letters are contiguous (nothing is between 'A' and 'B', for example) and the lowercase letters are contiguous. On these systems, there is some number  $x$  such that the uppercase and lowercase versions of a letter are always exactly  $x$  places apart. So, for example, `chr(ord('A') + x)` returns 'a', and `chr(ord('P') + x)` returns 'p'. Simi-

---

larly, a program can compute uppercase letters from lowercase letters using minus  $x$ . The number  $x$  can be computed using the relation

$$x = \text{ord}('a') - \text{ord}('A')$$

*nonprintable  
characters*

Another use for the `chr` function has to do with nonprintable characters. A computer frequently has the ability to send messages to the video screen that mean things like “clear the screen,” “ring the bell,” or some other manipulation of the output device other than simply writing a letter on the screen. These signals are usually considered to be values of type `char`. So, for example, if the “character” that rings the bell happens to be number 7 on a given type of terminal, then the following will ring the bell on that terminal:

```
write(chr(7))
```

*portability*

These sample uses of `chr` and `ord` are highly implementation-dependent. Different systems will order the values of type `character` differently. Even the list of available characters will differ from system to system. Some systems do not have a bell to ring. Some do not have the curly brackets `'{'` and `'}'`. Some do not have lowercase letters. Hence, any program written using these techniques will definitely not be portable. These sorts of manipulations should always be isolated into clearly documented procedures, so that they can be easily changed.

## TURBO Pascal

### Uppcase

In TURBO Pascal you can use the predefined function `upcase` to change letters from lowercase to uppercase. The function `upcase` takes one argument of type `char` and returns the corresponding uppercase version of the letter. So, for example, `upcase('a')` returns `'A'`. If there is no uppercase equivalent, then `upcase` returns its argument unchanged. In particular, if its argument is already uppercase, then it returns it unchanged. Hence, `upcase('n')` and `upcase('N')` both return `'N'`. This function can be used to test for a letter without having to test explicitly for both the uppercase and lowercase versions of the letter. For example, the following boolean expression will be true provided that the value of `Ans` is either `'n'` or `'N'`:

```
upcase(Ans) = 'N'
```

### Subrange Types

The simplest kind of type that can be defined within a Pascal program is a *subrange type*. These types are critically important in constructing other, more complicated and very useful types that we will introduce in the next chapter. They are also useful in their own right as an automatic error-checking facility. But before discussing their uses, we must describe what these types are.



A subrange type is obtained from an ordinal type by specifying two constants of that type. The type from which the two constants are chosen is called the *host type*. The values of the subrange type consist of the two constant values specified plus all the values of the host type that fall between the two specified constants. A subrange type is an ordinal type, and the values are ordered in the same way as they are in the host type. A subrange type definition is a declaration and so is placed in the declaration part of a program. For example, a program might start as follows:

*host  
type*

```
program Sample(input, output);
const PI = 3.14159;
type SmallInteger = -10 .. 10;
var Big: integer;
    Little: SmallInteger;
    X, Y: real;
```

The third line in this example is called a *type declaration*. The type `SmallInteger` is defined to be all `integer` values between `-10` and `+10`, including the endpoints. The variable `Little` is defined to be of type `SmallInteger`, and so it can take on values from `-10` to `+10`. Hence, the following is a Pascal statement that might legitimately appear in this program:

*type  
declaration*

```
Little := 4
```

However, the following should produce an error message when the program is run:

```
Little := 11{Not allowed.}
```

Subrange type declarations, as well as the other type declarations that we will introduce later, go between the constant and variable declarations. The general layout of the various declarations, is illustrated in Figure 8.12. The general form of a subrange type declaration follows the model of the sample shown in the figure: It consists of the identifier *type* followed by an identifier to serve as the name of the type, followed by an equal sign and then the definition of the type and a semicolon. The definition consists of two constants separated by two periods. The two constants are chosen from an ordinal type, and the first constant must be less than or equal to the second. Notice that defined constant names may be used for the constants in a subrange type definition, as shown in Figure 8.12.

You can define any number of subrange types. Each declaration has its own type name and type definition. The word *type* precedes the list of declarations. It is included only once, no matter how many subrange types are declared.

Subrange types and their host types are compatible in the sense that any value of a subrange type is also considered to be of the host type. For example, using the preceding program opening, any value of type `SmallInteger` is also of type `integer`. In this program, therefore, the following is a legitimate statement:

*type  
compatibility*

```
Big := Little
```

The following program code is also legitimate:

```
readln(Big);
Little := Big
```





With the variable `Final` declared to be of a subrange type as shown, the computer should give an error message if for some reason the value of `Final` is set to, say, 'G'. There is a limit to the amount of checking that can be handled by subrange types. Subrange types consist of *all* the values between the two limits. Hence, the above declaration does not provide for an error message in the event that the value of `Grade` is set to 'E'.

## TURBO Pascal

### The R Compiler Directive

TURBO Pascal does *not* always give an error message when the value of a subrange type is outside its defined range. However, you can ask it to give such error messages by inserting the following line in the file that contains your program:

```
{ $R+ }
```

It is a good idea to include this line while debugging your program. This line should come before your program and should not contain any blanks. This is called a *compiler directive* and will be discussed more fully in Chapter 9.

### The For and Case Statements Revisited

Ordinal types are like integers in the sense that they can be listed. Given two bounds, we can proceed from the lower bound to the next value in the ordered list, and then to the next, and so forth until we reach the upper bound. This is the only property of the type `integer` that the *for* statement uses. It thus seems natural to allow the loop control variable in a *for* statement to be of any ordinal type. Pascal does just that. Naturally, the expressions that give the initial and final values of the loop control variable must be of the same ordinal type as the variable. As an example, the following is a perfectly legitimate Pascal statement:

```
for I := 'a' to 'z' do  
    write(I)
```

Provided that `I` is declared to be of type `char` or an appropriate subrange type, this will output the alphabet.

Since a loop control variable assumes values between two bounds, it is possible to declare it as a subrange type. Moreover, this is a good idea, since it serves as one additional check on the program.

Now that we have defined the ordinal types, we can define the *case* statement more completely and compactly. The expression that governs a *case* statement may be of any ordinal type. The label lists must, of course, consist of constants of that ordinal type.

---

## Allowable Parameter Types

Procedures and functions may have parameters of any type whatsoever, including all the types we have seen so far and all the defined types we will introduce in succeeding chapters. However, a procedure or function declaration must use type names. It cannot use type definitions. The following is thus an illegal procedure heading:

```
procedure WriteGrades (Quiz1, Quiz2: 0 .. 100);  
{NOT ALLOWED}
```

Instead, you must declare a name for the defined type and use the name in the formal parameter list, like so:

```
type Score = 0 .. 100;  
procedure WriteGrades (Quiz1, Quiz2: Score);
```

*type returned  
by a function*

The same rule applies to the type returned by a function. You must use a type name, not a type definition, in the function declaration. Although there are no restrictions as to what types may be used as parameters (arguments) to functions, in the next chapter we will see that there are restrictions on what types may be returned. All the types we have seen thus far are allowed as the type of the value returned by a function.

---

## TURBO Pascal

---

### String Parameters

Since procedures and functions may have parameters of any type, they may have parameters of *string* types. However, every string type must be declared in a type declaration. If you want a procedure with a parameter of type *string*[10], then you cannot use the following procedure heading:

```
procedure ReadNames (var First, Last: string[10]);  
{NOT ALLOWED}
```

Instead, you must first define a type name and then use the type name, as follows:

```
type String10 = string[10];  
procedure ReadNames (var First, Last: String10);
```

This may seem quite silly, since you can even use a name that looks very much like *string*[10]. Nonetheless, the compiler will insist on this detail because *string*[10], *string*[15], and so forth are considered to be *type definitions*, much like *1 .. 10*; they are not considered to be *type names*, like *integer* and *char*.

In TURBO Pascal, you can have programmer-defined functions which return a value of a *string* type. For such function declarations, you must first define a type name

---



for the *string* type and then use that type name to specify the value returned. The situation is similar to what we described for parameters of *string* types and is illustrated in the following:

```
type String10 = string[10];
   String21 = string[21];
   . . .
function FullName (First, Last: String10): String21;
begin {Full Name}
  FullName := First + ' ' + Last
end; {FullName}
```

To aid you in designing functions and procedures for string handling, TURBO Pascal provides a number of predefined functions and procedures for manipulating strings. These are described later in this chapter.

## Pitfall

### Parameter Type Conflicts

Extra care must be taken when you are using a subrange type as the type of a variable parameter in a procedure. The rules for matching parameters state that the formal and actual parameters must be of the same type. This can sometimes cause subtle problems. As an example, recall the procedure *Exchange* defined in Figure 5.1. All the information we need about the procedure is given in the procedure heading. It as well as some other declarations that might appear in a program are

```
type Rating = 0 . . 10;
var A, B: Rating;
procedure Exchange (var X, Y: integer);
  {Interchanges the values of X and Y.}
```

In a program with these declarations, it seems perfectly natural to include a procedure call such as the following:

```
Exchange (A, B)
```

Although it seems natural, it will produce an error message and will prevent the program from running to completion. The reason is that the formal and actual parameter types do not match. The formal parameters are of type *integer*, whereas the actual parameters are of type *Rating*. Sometimes this problem can make the use of a subrange type impractical.

Type matching of value parameters is more complicated to explain, but it causes fewer problems because it allows a certain amount of mixing of types. The basic rule that applies for value parameters is the same as that for variable parameters: The formal and actual parameters must be of compatible types. In

fact, the value of the actual parameter must be of the same type as the formal parameter, but because of automatic type conversion and the overlapping of type values, this is a fairly forgiving rule.

A value of a subrange type is also considered to be a value of the host type, and so type conflicts with value parameters are not as likely to occur as they are with variable parameters. For example, consider the following procedure:

```
procedure WriteResult(X: integer);
begin
  writeln('The result is ', X)
end;
```

With A declared to be of type Rating, the following procedure call is perfectly valid:

```
WriteResult(A)
```

There is no type conflict in this case because it is the *value* of the actual parameter A that must agree in type with the formal parameter X. Although A is of type Rating, the *value* of A is considered to be of type integer, in addition to being of type Rating.

As has been noted in previous chapters, a value of type integer can be used anywhere that a value of type real can be used. The computer automatically converts the integer value to an approximately equivalent real value. Hence, if a formal value parameter is of type real, the actual parameter is allowed to be of type integer.

## Pitfall

### Anonymous Types

It is possible to use a subrange type definition directly in a variable declaration, rather than first defining a type name. For example, the following declares the variable I to be of type 1 . . . 10:

```
var I: 1 . . 10;
```

These unnamed types are called *anonymous types*, and it is poor programming practice to use them.

To understand the problem with anonymous types, you need to know how the Pascal language determines type equivalence. In Pascal, two types are not the same unless they have the same name. As an absurd but very illustrative example, consider the type declarations below:

```
type SmallInteger = 1 . . 10;
      PetiteInteger = 1 . . 10;
```

*type  
equivalence*

In Pascal the two types `SmallInteger` and `PetiteInteger` are considered to be *different types*! Hence, among other things, an actual *variable parameter* of type `SmallInteger` cannot be substituted for a formal *variable parameter* of type `PetiteInteger` in a procedure call.

A more likely instance of this problem involves anonymous types. Consider the following declarations:

```
type SmallInteger = 1 . . 10;  
var X: 1 . . 10;
```

With these declarations, the variable `X` cannot be used as an actual *variable parameter* of type `SmallInteger`. Since the type of the variable `X` has no name whatsoever, it cannot have the same name as any named type. Hence, it cannot be an actual variable parameter in any procedure whatsoever! (It can be an actual *value parameter*, but that is not much consolation.)

The way to avoid these problems is obvious: Always declare a unique name for every defined type. This lesson applies to all programmer-defined types, including the ones we will introduce in the chapters that follow.

---

## Case Study

---

### Complete Calendar Program

#### Problem Definition

We want to embed the calendar display procedure from Figure 7.19 in a complete calendar program. The calendar program will accept a month and year which are input as two integers and will then output the calendar display for that particular month and year.

Calendars are now taken for granted, and so we tend to forget that they are a complicated computational task. Historically they have presented society with serious problems, ones that have not always been completely solved. The calendar design has changed more than once. A date like January 1, 1132, has more than one possible interpretation, since a different calendar was used then as opposed to now (which makes one wonder what time the word “then” refers to). As we indicated in Chapter 5, calculating leap years is complicated enough to require a special computation for some century years. Moreover, if we go back far enough in history or go forward far enough into the future, then that algorithm will not work accurately enough, and we will need to complicate the calculation even more. To eliminate all these problems, we will only require that the program apply to years in the relatively near past and the relatively near future. Specifically, we will only require that the program work for the years 1901 through 2999. For those years, a year is a leap year if it is divisible by 4, and no additional corrections are needed. This rule even works correctly for the century year 2000, although it does not work correctly for the years 1900 and 3000.

*refining  
the definition*



## Discussion

As was indicated in the previous paragraph this task is likely to be computationally complicated and delicate. A hastily constructed solution will undoubtedly give incorrect results as well as the frustration that results from wasted effort. This problem will require more care and more inspiration than the other problems we have seen, and so we will proceed slowly and methodically.

The task of displaying the calendar for a given month can be divided into three main subtasks:

DayOne: Determine what day of the week (Sunday, Monday, etc.) the first day of the month falls on.

TotalDays: Determine how many days there are in the month.

DisplayMonth: Call the procedure given in Figure 7.19.

*use of  
functions*

The data flow diagram for this decomposition is given in the top portion of Figure 8.13. Since the subtasks DayOne and TotalDays each do nothing other than produce a single value, they will be implemented as functions. To compute the number of days in month 2 (February), the procedure TotalDays will need to determine whether the year is a leap year or not. To find out, it will call a boolean-valued function called LeapYear, as shown in the data flow diagram.

*subrange  
types*

The only meaningful values for the variable Month are the numbers 1 through 12. Hence, we can declare it to be of the subrange type 1 . . 12. That way, if we make a mistake and allow the program to set it equal to some other value, we will get an error message that will help us to locate the mistake. We will call this subrange type MonthInteger. Similarly, the variable Year can be declared to be of the subrange type 1901 . . 2999, which we will name YearInteger.

The function TotalDays will return the number of days in a month. We could use the type integer as the type of the value returned, but it would be preferable to use a subrange type as an additional check for mistakes in the function computation. We will therefore define a type MonthDay as 1 . . 31 and use this as the type of the value returned by the function. Thus, our program will include the following declarations (and possibly others as well):

```
type MonthDay = 1 . . 31;
    MonthInteger = 1 . . 12;
    YearInteger = 1901 . . 2999;
var Month: MonthInteger;
    Year: YearInteger;
```

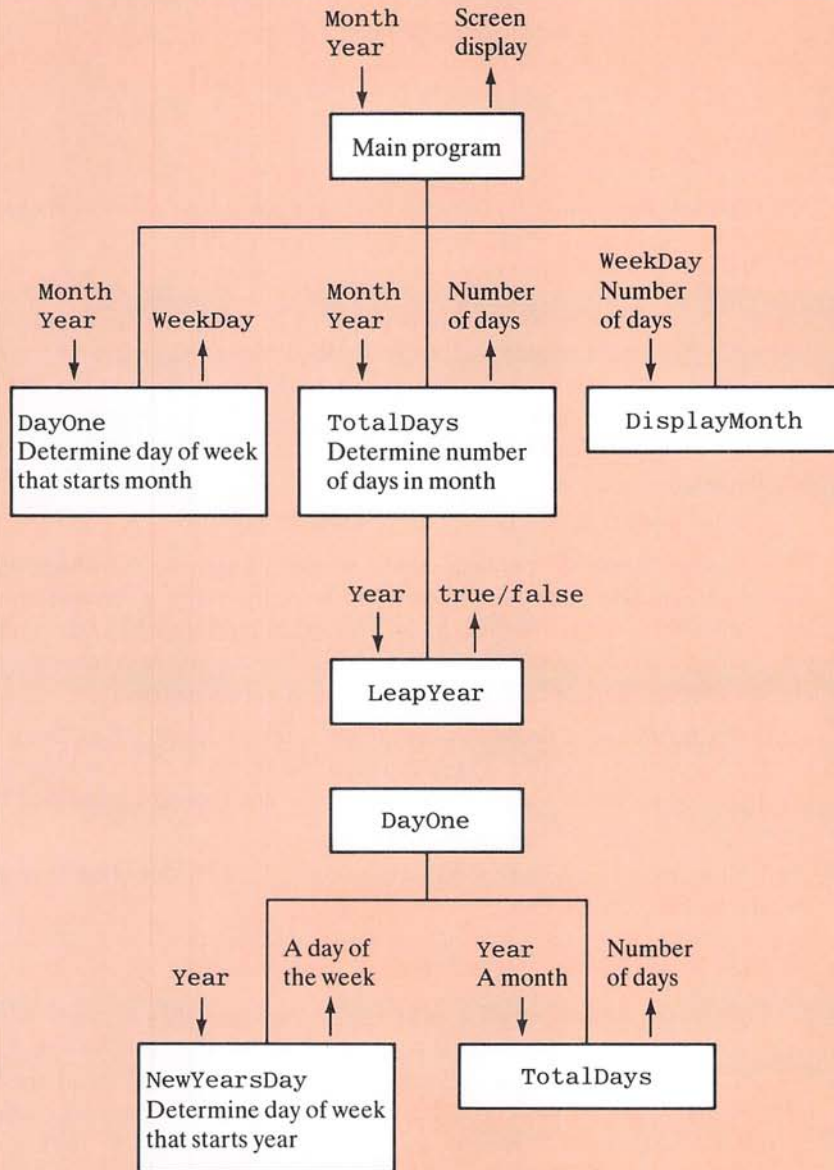
**ALGORITHM**  
*for TotalDays*

The function TotalDays is one part of the program that is easy to construct. The procedure heading shows the subrange type that is returned:

```
function TotalDays (Month: MonthInteger,
                    Year: YearInteger): MonthDay;
```

The computation is carried out by a single case statement:

```
case Month of
  4, 6, 9, 11: TotalDays := 30;
```



### Data Summary

Month: an integer representing the month. (1 for January, 2 for February, etc.)

Year: the year. (Assumed to be in the range 1901 through 2999. 1901 and 2999 are permitted.)

WeekDay: an integer representing the first day of the week for the specified month and year.

(0 for Sunday, 1 for Monday, . . . , 6 for Saturday.)

**Figure 8.13**  
Data flow  
diagram  
for calendar  
program.

```

1, 3, 5, 7, 8, 10, 12: TotalDays := 31;
2: if not LeapYear(Year) then
    TotalDays := 28
else
    TotalDays := 29
end

```

The complete function declaration is trivial to produce by combining these two items.

### Discussion of DayOne

The function DayOne is the complicated part of this programming assignment. It takes two arguments, Month and Year, and should return the day of the week (Sunday, Monday, etc.) for the start of the given month of the specified year. The days of the week are coded as the integers of a subrange type called WeekDay, which we define to be 0 . . 6. (Sunday is 0, Monday is 1, etc.) We thus need to add another type declaration:

```
type WeekDay = 0 . . 6; {Sunday is 0.}
```

January 1,  
1901

Our approach to this problem will be to take a fixed date for which we know the day of the week and to calculate days of the week by counting forward from that day. January 1, 1901, was a Tuesday. Since Tuesday is represented by the number 2, we will define a constant called JanFirst1901 to be 2. The calculation for any values of the arguments Month and Year will proceed in two stages:

1. NewYearsDay: Calculate the day of the week for January 1 of Year.
2. DayOne: Use the result of task 1 to calculate the day of the week for the first day of the Month in that Year. (This will require knowing how many days there are in each month from January up to Month.)

The bottom part of the data flow diagram in Figure 8.13 fits this breakdown of subtasks into the overall computation.

### Discussion of NewYearsDay

To obtain the algorithm for task 1, note that all years are either 365 days long or, in the case of leap years, 366 days long. If you look at the calendars for two successive years, you will see that if a year has 365 days, then January 1 moves ahead one day of the week the next year. (If you do not have two calendars handy, you can confirm this by the fact that  $(365 \bmod 7)$  is equal to 1.) Hence, since January 1, 1901, is a Tuesday (day 2), January 1, 1902, must be a Wednesday (day 3), and January 1, 1903, must be a Thursday (day 4). Although 1904 is a leap year, January 1, 1904, occurs before the extra leap-year day is inserted, and so it is a Friday (day 5). January 1, 1905, falls after the extra day inserted in February 1904, and so January 1, 1905, falls on a Sunday (day 0, obtained as  $(5 + 2) \bmod 7$ ).

To obtain the day of the week for any January first after 1901, we can start with the day JanFirst1901, add one for each 365-day year, and add two for each leap year up to the desired year. All this addition is performed counting by sevens, so that 7 is the same as 0; in other words, we apply the operator  $\bmod 7$  to our calculations.



Hence, our first try at an algorithm for the function `NewYearsDay` is to return the value

```
(JanFirst1901 + (number of non-leap years) + 2 * (the number of leap years) ) mod 7
```

An alternative and easier algorithm is to add one for *every* year and then add only 1 instead of 2 for the leap years. This leads to the following code:

```
ElapsedYears := Year - 1901;
LeapYearCount := (ElapsedYears div 4);
return the value
(JanFirst1901 + ElapsedYears + LeapYearCount) mod 7
```

*ALGORITHM  
for NewYearsDay*

## Discussion of DayOne, Concluded

All that remains is to construct the algorithm for the function `DayOne`. The function has two arguments, one for the `Month` and one for the `Year`. The computation starts with the value

```
NewYearsDay (Year)
```

and then adds in the number of days in every month up to `Month` to obtain an integer called `DayCount`. The day of the week for the first day of the specified month is then computed as

```
DayCount mod 7
```

The complete program, including this function, is given in Figure 8.14.

---

## Use of Subrange Types for Error Detection

A common mistake when calling a function or procedure is to give the parameters in an incorrect order. For example, if the function `DayOne` is declared as in Figure 8.14, then the following will evaluate to the first day of February 1950:

```
DayOne (2, 1950)
```

Suppose that we accidentally reverse the arguments:

```
DayOne (1950, 2)
```

Since we used subrange types for the formal parameters, the computer will detect an error and output an error message. The computer will see that the value 1950 is not of the type `MonthInteger`, which is defined as 1 . . . 12, and will alert you to a problem. If the parameters had instead been defined to be of type `integer`, then the function might simply return an incorrect value. Such a mistake is likely to go unnoticed if you do not use subrange types. (Do you know what day of the week February 1, 1950, fell on?) Even if the computer does catch this mistake in some other way, it is not likely to find the correct location of the mistake unless you use subrange types.

---

**Program**

```

program Calendar(input, output);
{Displays a calendar for any month from January 1901 through December 2999.}
type WeekDay = 0 . . 6; {Sunday is 0.}
    MonthDay = 1 . . 31;
    MonthInteger = 1 . . 12;
    YearInteger = 1901 . . 2999; {The simple
    leap-year calculation used does not work for 1900 or 3000.}
var Month: MonthInteger;
    Year: YearInteger;
    Ans: char;

function NewYearsDay(Year: YearInteger): WeekDay;
{Returns a code for the day of the week for January 1 of the specified Year.
The function does not work correctly if you change the type of
Year to integer and use years such as 1900 or 3000.}
const JanFirst1901 = 2; {Tuesday}
var ElapsedYears: integer;
    LeapYearCount: integer;
begin{NewYearsDay}
    ElapsedYears := Year - 1901;
    LeapYearCount := (ElapsedYears div 4);
    NewYearsDay :=
        (JanFirst1901 + ElapsedYears + LeapYearCount) mod 7
end; {NewYearsDay}

function LeapYear(Year: YearInteger): boolean;
{Returns true if Year is a leap year. This function will not work
correctly if the type of Year is changed to the type integer. For example,
1900 and 3000 would both return true, but neither one is a leap year.}
begin{LeapYear}
    LeapYear := (Year mod 4) = 0
end; {LeapYear}

function TotalDays(Month: MonthInteger;
    Year: YearInteger): MonthDay;
{Returns the number of days in the Month of the given Year.}
begin{TotalDays}
    case Month of
        4, 6, 9, 11: TotalDays := 30;
        1, 3, 5, 7, 8, 10, 12: TotalDays := 31;
        2: if not LeapYear(Year) then
            TotalDays := 28
        else
            TotalDays := 29
    end {case}
end; {TotalDays}

```

**Figure 8.14**  
Complete calendar  
program.

```

function DayOne (Month: MonthInteger;
                 Year: YearInteger): WeekDay;
{Returns a code for the day of the week for the first day of the Month
of the specified Year. Calls the functions NewYearsDay and TotalDays.}
var DayCount: integer;
    PastMonth: MonthInteger;
begin{DayOne}
    DayCount := NewYearsDay (Year);
    for PastMonth := 1 to (Month - 1) do
        DayCount := DayCount + TotalDays (PastMonth, Year);
    DayOne := DayCount mod 7
end; {DayOne}

procedure DisplayMonth (NumberOfDays: MonthDay; FirstDay: WeekDay);
{Displays the usual layout for a month with NumberOfDays days in it.
FirstDay codes the first day of the month: 0 for Sunday, 1 for Monday, etc.
The procedure code is identical to that in Figure 7.19, but the parameter
types have been changed to subrange types in order to provide an additional
check for errors in the program.}
const Width = 4; {Field width for one day of the calendar.}
    Blank = ' ';
var DayCount, CountAll: integer;
begin{DisplayMonth}
    writeln('Sun':Width, 'Mon':Width,
           'Tue':Width, 'Wed':Width,
           'Thu':Width, 'Fri':Width, 'Sat':Width);
    for CountAll := 0 to FirstDay - 1 do
        write(Blank :Width);
    CountAll := FirstDay;
    for DayCount := 1 to NumberOfDays do
        begin{for}
            CountAll := CountAll + 1;
            {If (CountAll mod 7) = 0, then day number DayCount is a Saturday.}
            write(DayCount :Width);
            if (CountAll mod 7) = 0 then
                writeln
            end; {for}
        writeln
    end; {DisplayMonth}

begin{Program}
    writeln('This program will display the calendar for');
    writeln('any month from the years 1901 to 2999.');
```

**Figure 8.14**  
(continued)



```

repeat
    writeln('Enter month and year as two integers:');
    readln(Month, Year);
    writeln('Month ' :12, Month :2, ' Year ', Year :4);
    DisplayMonth(TotalDays(Month, Year), DayOne(Month, Year));
    writeln('Do you want to see another month? (y/n)');
    readln(Ans)
until (Ans = 'n') or (Ans = 'N');
writeln('Have a good month!')
end. {Program}

```

### Sample Dialogue

This program will display the calendar for any month from the years 1901 to 2999.

Enter month and year as two integers:

6 1944

Month 6 Year 1944						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Do you want to see another month? (y/n):

no

Have a good month!

**Figure 8.14**  
(continued)

## Enumerated Types (Optional)

There is one other kind of ordinal type. Types of this kind are called *enumerated types*. An enumerated type is just a list of values named by identifiers. Enumerated types have some similarity to “society types” who have lost their wealth. The values of an enumerated type have no properties other than their order and their names. Despite their simple nature, enumerated types can be useful. By choosing the identifiers to be meaningful names, you can sometimes use enumerated types to make a program easier to read. For example, the following type declaration declares the names of four kinds of vehicles to be values of an enumerated type:

```
type Vehicle = (Motorcycle, Car, Bus, Truck);
```

This type has four values named by the four identifiers in the list.

*declaration*

All type declarations are given in the same place in a program, in the same manner that we described for subrange types. Moreover, it is permissible to intermix subrange, enumerated, and other, yet to be introduced, type declarations. The identifier *type* is

written once, and then each identifier in the type declaration section is set equal to its definition. For an enumerated type, the type definition follows the above example. The definition consists of a list of identifiers enclosed in parentheses. The list of identifiers are the names of the constants of that type, and they are ordered as in the list.

As with other types, there can be variables of an enumerated type. For example, the following declares `Class` to be of type `Vehicle`:

```
var Class: Vehicle;
```

Variables of an enumerated type and their values behave much like those of any other type. For example, the following makes `Bus` the value of the variable `Class`:

```
Class := Bus
```

One common use of enumerated types is in `case` statements, such as the following:

```
case Class of
  Motorcycle: Toll := 0.25;
  Car: Toll := 0.50;
  Truck, Bus: Toll := 1.00
end
```

The enumerated type serves two purposes here: It makes the program meaning clearer, and it guarantees that the `case` statement is almost always defined. The variable `Class` cannot take on a value that is not on some statement label list. The only way that the `case` statement can be undefined is if the variable `Class` were never initialized. Of course, a subrange type can sometimes serve the same purpose, but an enumerated type allows more flexibility in choosing label names.

It is important to note that the elements of an enumerated type are not strings and can neither be read in nor written out by a program. In a program with the above declarations, the following will produce no output other than an error message:

```
write(Truck) {Not allowed.}
```

If you wish to output a value of an enumerated type, you must somehow output a string value that indicates the value of the variable. For example, the following will output the value of the variable `Class` of type `Vehicle`:

```
case Class of
  Motorcycle: write('Motorcycle');
  Car: write('Car');
  Bus: write('Bus');
  Truck: write('Truck')
end {case}
```

This may seem like a lot of work just to add two quotes, but it is necessary. The values `Truck` and `'Truck'` are very different; they are not even of the same data type.

Figure 8.15 shows a program that uses an enumerated type for the days of the week. Notice that since an enumerated type is an ordinal type, it can be the type of a `for` loop control variable.

*variables*

*uses of  
enumerated  
types*

**Program**

```

program Payroll(input, output);
{Computes an hourly employee's weekly pay.}
const Width = 8; {Field width for total wages.}
      SatAdjustment = 1.5; {Time and a half.}
      SunAdjustment = 2.0; {Double time.}
type WeekDay = (Mon, Tue, Wed, Thur, Fri, Sat, Sun);
var BaseRate, Wages: real;
    Day: WeekDay;
    Hours: integer;

function Rate(BaseRate: real; Day: WeekDay): real;
{Returns the rate of pay for the day of the week.}
begin{Rate}
  case Day of
    Mon, Tue, Wed, Thur, Fri: Rate := BaseRate;
    Sat: Rate := SatAdjustment*BaseRate;
    Sun: Rate := SunAdjustment*BaseRate
  end {case}
end; {Rate}

begin{Program}
  writeln('Enter the basic hourly wage rate:');
  readln(BaseRate);
  Wages := 0.0;
  writeln('Enter the hours worked');
  writeln('for Monday through Sunday:');
  for Day := Mon to Sun do
    begin{for}
      read(Hours);
      Wages := Wages + Hours*Rate(BaseRate, Day)
    end; {for}
  readln;

  writeln('Wages for the week total: $', Wages :Width:2)
end. {Program}

```

**Sample Dialogue**

```

Enter the basic hourly wage rate:
10.00
Enter the hours worked
for Monday through Sunday:
8 8 8 8 0 2
Wages for the week total: $ 440.00

```

**Figure 8.15**  
(Optional)

**Program using an  
enumerated type.**



## TURBO Pascal

### String Functions and Procedures

In this section we will describe some of the predefined TURBO Pascal functions and procedures for string manipulation. A complete list of them is given in the summary at the end of this chapter. These functions and procedures are not available in standard Pascal. Hence, programs written using them are not completely portable. Nonetheless, they are extremely useful for programs that you know will only be used on TURBO Pascal systems.

The TURBO Pascal predefined string functions and procedures do not discriminate among the various *string* types, such as *string*[10] and *string*[25]. When they have a *string* variable parameter, the string variable used in the procedure call may be of any *string* type.

Frequently, one wants to search for a string and replace one particular word (or other substring) with another. In TURBO Pascal this is particularly easy. The function *pos* can be used to locate the word which is to be replaced. For example, suppose that *Line* is a variable of a *string* type that contains a line of text. Suppose that we want some Pascal code that will replace the word 'hard' in that line with the word 'easy'.

We first must locate the word 'hard'. The function *pos* is designed for just this purpose. It has two *string* arguments and returns an integer telling us where its first argument occurs within its second argument. For example, consider the statement

```
N := pos('hard', Line)
```

This function *pos* returns an integer value that gives the position of the first letter of the string 'hard' within the value of *Line*. If 'hard' is the first word in *Line*, it returns 1, setting the value of the integer variable *N* equal to 1. If the first letter of 'hard' is the tenth character in *Line*, it returns 10. If 'hard' does not occur at all within the string value of *Line*, then *pos* returns 0.

Notice that *pos* can be used to test for the presence or absence of a pattern within a string. For example, the following boolean expression will be true if and only if 'hard' occurs somewhere within the string value of *Line*:

```
pos('hard', Line) > 0
```

Again, suppose that *Line* is a variable of a *string* type and suppose that the program contains

```
N := pos('hard', Line)
```

where *N* is of type integer and *Line* contains the string

```
'It is hard to do text editing.'
```

The value of *N* will then be set to 7. Note that the blank is counted as a symbol.

*pos*

*delete*

The word 'hard' can be removed using the procedure `delete`. The general form of a procedure call to `delete` is

```
delete(<string var>, <start>, <number>)
```

<string var> is a variable parameter of type *string*; <start> and <number> are both value parameters of type *integer*. This call will delete <number> characters from <string var>, starting at position <start>. For example, to delete the word 'hard' from the string in the variable `Line` of our example, the following will work:

```
delete(Line, N, 4)
```

*insert*

To complete this example, we use the TURBO Pascal procedure `insert`. The general form of a call to this procedure is

```
insert(<new piece>, <string var>, <start>)
```

<new piece> is a value parameter of type *string*, <string var> is a variable parameter of type *string*, and <start> is a value parameter of type *integer*. This procedure call will insert the string <new piece> into the string in the variable <string var> starting at position <start>. To complete our example, the following will change the value of `Line` to what we want:

```
insert('easy', Line, N)
```

After this procedure call, the value of `Line` will be

```
'It is easy to do text editing.'
```

*copy*

Two other useful string editing functions are `length` and `copy`. `length` has one string argument and returns the length of that string. For example,

```
N := length('the hat')
```

sets the integer variable `N` equal to 7.

The function `copy` is a bit more complicated to describe. It can be used to "copy out" a substring of a given string. A call to `copy` is of the form

```
copy(<string>, <start>, <length>)
```

where <string> is an expression of type *string* and the remaining two arguments are expressions of type *integer*. The value returned is the substring of <string> starting with symbol number <start> and continuing for <length> symbols. For example, suppose that `Line` has the string value

```
'It is easy to do text editing.'
```

and consider the following statement:

```
writeln(copy(Line, 7, 10))
```

The call to `copy` returns the string

```
'easy to do'
```

and so this `writeln` statement causes the following to be written to the screen:

```
easy to do
```

---

## TURBO Pascal Case Study

### Replacing Substrings

#### Problem Definition

The government is attempting to train its employees to avoid sexist overtones in letters and other official documents. It has hired you to write a program that will take as input any sentence and then rewrite it using gender neutral terms. A sentence such as “A lawyer and his client must be allowed contact.” should be rewritten to read “A lawyer and her or his client must be allowed contact.”

#### Discussion

Much sexist language, of the male chauvinist type, can be removed from a sentence by replacing male gender pronouns like “he” and “him” with gender neutral words or phrases like “she or he” and “her or him.” That is the approach we will use. Our program will use the TURBO Pascal string-manipulating functions and procedures to replace male pronouns with gender neutral terms. To this end, we will define a new string-manipulating procedure that takes a string variable and changes its value by replacing all occurrences of a given subpattern with some given substitution string. For example, the procedure could be used to replace all occurrences of ‘he’ with ‘she or he’. The procedure heading, together with the needed type declaration, will be as follows:

```
type StringType = string[80];
procedure ReplaceAll (Old, New: StringType; var Source: StringType);
{Changes the value of Source by replacing all occurrences of
the string Old with the string New.}
```

The technique for replacing one string with another was illustrated in the previous section when we introduced the function and procedures `pos`, `delete`, and `insert`. Using this technique we can replace the first occurrence of the string `Old` with the string `New` using the following algorithm:

*ALGORITHM  
for substring  
replacement*

1. Use `pos` to find the starting position of `Old` in `Source`; store this position in the integer variable `Position`.
2. Use `delete` to delete `length(Old)` symbols from `Source` starting at symbol number `Position`.
3. Use `insert` to insert the string `New` at position number `Position`.

Converting this algorithm to Pascal code produces

```
begin {Replace one occurrence of Old with New.}
  Position := pos(Old, Source);
  delete(Source, Position, length(Old));
  insert(New, Source, Position)
end {Replace one occurrence of Old with New.}
```



In order to replace all occurrences of Old with New, we simply repeat this as long as Source contains an occurrence of Old. If Position is set as in the above loop body, then we repeat the loop provided that Position is greater than zero. Since it is possible that Old does not occur at all in Source, we will use a *while* loop to allow for the possibility of iterating the loop zero times. This requires that Position be set before the loop begins. This, in turn, requires rewriting the loop body to yield the following:

```
Position := pos(Old, Source);
while Position > 0 do
  begin {Replace one occurrence of Old with New.}
    delete(Source, Position, length(Old));
    insert(New, Source, Position);
    Position := pos(Old, Source)
  end {Replace one occurrence of Old with New.}
```

One way to arrive at this *while* loop is to use the “unrolling a loop” technique discussed in Chapter 7.

*inadvertent  
replacements*

If you test the above algorithm in a program or if you test it using pencil and paper, you will find that it can result in an infinite loop. Consider the sequence

'I saw him.'

and apply the algorithm to replace 'him' with

'her or him'

The algorithm replaces 'him' to produce the string

'I saw her or him.'

So far so good, but it then continues to replace 'him' again to produce

'I saw her or her or him.'

Since there is always an occurrence of 'him' in the string, the replacement loop will never terminate.

To eliminate the possibility of such infinite loops, our program will first replace all occurrences of the string Old (e.g., 'him') with a special string Marker that does not cause an infinite loop (e.g., Marker = 'xxx'). The program will then go on to replace Marker with the string New (e.g., 'her or him').

Following the plan of the previous paragraph, we will first define a procedure SafeReplace that will implement the problematic replacement algorithm described above. Although it cannot perform an arbitrary string replacement, it will correctly carry out certain safe replacements, such as replacing all occurrences of Old with Marker. Once we have this restricted procedure SafeReplace, any string Old can be replaced by any string New using the two procedure calls

```
SafeReplace(Old, Marker, Source);
SafeReplace(Marker, New, Source)
```

We can now write our general procedure `ReplaceAll`. It will simply consist of the above two calls to the procedure `SafeReplace`. The procedures `SafeReplace` and `ReplaceAll` are shown in Figure 8.16. You may wish to study them (but not the rest of Figure 8.16) before reading our derivation of the rest of this program.

Using the special symbol `Marker`, we were able to eliminate one kind of inadvertent substitution, but our problems are not yet over. There are other types of inadvertent substitutions. Suppose that `Source` contains the string

```
'The lawyer said he was innocent.'
```

We would hope to see it changed to

```
'The lawyer said she or he was innocent.'
```

However, the procedure call

```
ReplaceAll('he', 'she or he', Source)
```

will change the value of `Source` to

```
'Tshe or he lawyer said she or he was innocent.'
```

The problem is that 'he' occurs in 'The'. Hence, we must replace only *whole words* and not subwords, as the procedure `ReplaceAll` does.

We define a procedure `ReplaceWords` that is like `ReplaceAll` except that it replaces only whole words. This procedure will replace not just the word but the word surrounded by blanks (or other delimiters). The replacements are accomplished by calls to `ReplaceAll`, such as

```
ReplaceAll (<blank>+'he'+<blank>,  
           <blank>+'she or he'+<blank>, Source)
```

The procedure `ReplaceWords`, shown in Figure 8.16, uses `concat` rather than `+` to provide practice in using this new function, but `+` would have worked equally well.

The procedure also uses other delimiters besides blanks to find word endings, but the basic technique is the same in all cases. To check for initial words, the procedure assumes that sentences start with capital letters. For example, it searches for 'He' followed by a blank in order to find an initial 'he'. The complete program is given in Figure 8.16.

---

It would produce a dreadful mess if  
we were to do anything together.  
You see, we're different types.

*Overheard at a cocktail party*

---

*more inadvertent  
substitutions*

**Program**

```
program NonSexistDrill;  
  {Changes male pronouns to gender neutral terms.}  
  const LineLength = 80;  
  type StringType = string[LineLength];  
  var Sentence: StringType;  
      Ans: char;  
  
  procedure Instructions;  
  begin {Instructions}  
    writeln('This program will take one line sentences');  
    writeln('and make them less sexist by reducing the');  
    writeln('reliance on masculine pronouns. ');  
    writeln  
  end; {Instructions}  
  
  procedure SafeReplace(Old, New: StringType; var Source: StringType);  
  {Changes the value of Source by replacing all occurrences of  
  the string Old with the string New. Precondition: Old is not  
  a substring of New (or of Source with Old replaced by New).}  
  var SizeOld, Position: integer;  
  begin {SafeReplace}  
    SizeOld := length(Old);  
    Position := pos(Old, Source);  
    while Position > 0 do  
      begin {Replace one occurrence of Old with New.}  
        delete(Source, Position, SizeOld);  
        insert(New, Source, Position);  
        Position := pos(Old, Source)  
      end {Replace one occurrence of Old with New.}  
    end; {SafeReplace}  
  
  procedure ReplaceAll(Old, New: StringType; var Source: StringType);  
  {Changes the value of Source by replacing all occurrences of  
  the string Old with the string New. Precondition: The string Marker  
  (see const declaration) does not occur in Old, New, or Source.}  
  const Marker = 'xxx'; {A string that is not expected to occur elsewhere.}  
  begin {ReplaceAll}  
    SafeReplace(Old, Marker, Source);  
    SafeReplace(Marker, New, Source)  
  end; {ReplaceAll}
```

**Figure 8.16**

**String-  
manipulating  
TURBO Pascal  
program.**



```

function Cap(Word: StringType): StringType;
{Returns Word with its first character changed to uppercase.}
var Letter: char;
    EndWord: StringType;
begin{Cap}
    Letter := upcase(Word[1]);
    EndWord := Word;
    delete(EndWord, 1, 1);
    Cap := concat(Letter, EndWord)
end; {Cap}

procedure ReplaceWords(Old, New: StringType; var Source: StringType);
{Changes the value of Source by replacing all occurrences of
the word Old with the string New. Assumes that the beginning of a word
is marked by either capitalization or a blank. Assumes that the end
of a word is marked by a blank, comma, or period.}
const Blank = ' ';
    Comma = ',';
    Period = '.';
begin{ReplaceWords}
    ReplaceAll(concat(Blank, Old, Blank),
                concat(Blank, New, Blank), Source);
    ReplaceAll(concat(Blank, Old, Comma),
                concat(Blank, New, Comma), Source);
    ReplaceAll(concat(Blank, Old, Period),
                concat(Blank, New, Period), Source);

    ReplaceAll(concat(Cap(Old), Blank),
                concat(Cap(New), Blank), Source);
    ReplaceAll(concat(Cap(Old), Comma),
                concat(Cap(New), Comma), Source);
    ReplaceAll(concat(Cap(Old), Period),
                concat(Cap(New), Period), Source)
end; {ReplaceWords}

procedure NonSexists(var Sentence: StringType);
{Replace "he", "him", and "his" with gender neutral equivalents.}
begin{NonSexists}
    ReplaceWords('he', 'she or he', Sentence);
    ReplaceWords('him', 'her or him', Sentence);
    ReplaceWords('his', 'her(s) or his', Sentence)
end; {NonSexists}

```

**Figure 8.16**  
(continued)

```
begin{Program}  
  Instructions;  
  repeat  
    writeln('Enter a sentence: ');  
    readln(Sentence);  
    NonSexists(Sentence);  
    writeln('Try: ');  
    writeln(Sentence);  
    writeln('Again? (y/n) ');  
    readln(Ans)  
  until upcase(Ans) = 'N';  
  writeln('End of exercise. ')  
end. {Program}
```

### Sample Dialogue

This program will take one line sentences and make them less sexist by reducing the reliance on masculine pronouns.

Enter a sentence:

**He who eats nails is made of iron.**

Try:

She or he who eats nails is made of iron.

Again? (y/n)

**yes**

Enter a sentence:

**See an adviser, talk to him, and listen to him.**

Try:

See an adviser, talk to her or him, and listen to her or him.

Again? (y/n)

**yes**

Enter a sentence:

**We can use him if he is a real he-man.**

Try:

We can use her or him if she or he is a real he-man.

Again? (y/n)

**no**

End of exercise.

**Figure 8.16**  
(continued)

---

## Summary of Problem Solving and Programming Techniques

- When you are designing a program, it is very common for one or more subtasks to be the computation of a single value. In those cases, the subtasks should be implemented as functions.
-

- A restatement of a problem definition can often yield an algorithm or a hint of an algorithm for the problem.
- When you are designing functions, it is usually a bad idea to allow side effects, such as changing a global variable or using a variable parameter. If such things are needed, it is usually clearer to use a procedure rather than a function.
- A boolean-valued function can simplify an *if-then-else* statement by making a separate task of evaluating the boolean expression.
- When you are designing a function (or any other program part, such as a loop), it is wise to check for special cases and extreme values such as the first or last value in a list.
- Any ordinal type may be used as the type of the control variable of a *for* loop or as the type of the controlling expression of a *case* statement.
- Declaring variables to be of a subrange type is a form of error checking. If your program erroneously produces a value outside the expected range, the computer will detect this and issue an error message.

## TURBO Pascal

### Summary of String Functions and Procedures

#### String Functions

##### length

Syntax:

```
length(<string>)
```

Returns the number of occurrences of characters in <string>. <string> can be any string expression.

Example:

```
length('cool') returns 4
```

##### concat

Syntax:

```
concat(<argument list>)
```

Returns the result of concatenating all of the arguments in the order in which they occur. The <argument list> is any list of string expressions separated by commas. Note that the same effect can be obtained by using the operator +. For example,

```
'ab' + 'cd' + ' ef'
```

has the value 'abcd ef'.

Example

```
concat('ab', 'cd', ' ef') returns 'abcd ef'.
```



**pos**

Syntax:

```
pos (<pattern>, <target>)
```

Both arguments are string expressions. The value returned is an integer value giving the location of the first occurrence of <pattern> within <target>. A returned value of 1 means that <pattern> starts with the first character of <target>, 2 means the second character of <target>, and so forth. If <pattern> does not occur in <target>, then pos returns 0.

Examples:

```
pos('abc', 'abcd') returns 1
pos('bc', 'abcd') returns 2
pos('be', 'dobedobe') returns 3
pos('BE', 'dobedobe') returns 0
```

**copy**

Syntax:

```
copy (<string>, <start>, <length>)
```

<string> is a string expression, and the other two arguments are expressions of type integer. The value returned is the substring of <string> consisting of the <length> symbols starting at position <start>. Positions are numbered 1, 2, etc. The integer <start> must be in the range 1 . . 255. If <start> does not mark a position in <string>, then the empty string is returned. If the end of <string> is reached before <length> symbols are found, then copy does the best that it can and returns up to the end of <string>.

Examples:

```
copy('abcdef', 3, 2) returns 'cd'.
copy('Gloves are warm', 2, 4) returns 'love'.
copy('warm', 2, 10) returns 'arm'.
copy('Gloves', 20, 4) returns '', i.e., the empty string.
```

**upcase**

Syntax:

```
upcase (<char expression>)
```

Returns the uppercase version of <char expression>. <char expression> is a value parameter of type char. If there is no uppercase version of <char expression>, then it returns <char expression> unchanged.

Examples:

```
upcase('a') returns 'A'.
upcase('A') returns 'A'.
```

---

## String Procedures

### delete

Syntax:

```
delete (<string var>, <start>, <number>)
```

<string var> is a variable of a *string* type. <start> and <number> are integer expressions. The value of the variable <string var> is changed by deleting <number> symbols starting in position <start>. Positions are numbered 1, 2, etc. <start> must be in the range 1 . . 255. If <start> is greater than the length of <string var>, no symbols are removed. If the end of the value of <string var> is reached before <length> symbols are deleted, then copy does the best that it can and deletes up to the end of the string in <string var>.

Examples:

```
X := 'abcdef'; delete(X, 3, 2); writeln(X)
```

produces the output

```
abef
```

```
X := 'Gloves are warm'; delete(X, 2, 4); writeln(X)
```

produces the output

```
Gs are warm
```

```
X := 'warm'; delete(X, 2, 10); writeln(X)
```

produces the output

```
w
```

```
X := 'Gloves'; delete(X, 20, 4); writeln(X)
```

produces the output

```
Gloves
```

### insert

Syntax:

```
insert (<new piece>, <string var>, <start>)
```

<new piece> is a string expression. <string var> is a variable of a *string* type. <start> is an integer expression. The value of the variable <string var> is changed by inserting <new piece> into the value of <string var> starting in position <start>. Positions are numbered 1, 2, etc. <start> must be in the range 1 . . 255. If <start> is greater than the length of the string in <string var>, then <new piece> is concatenated on the end. If the result would be longer than the declared maximum length of <string var>, then any symbols over this length are truncated so <string var> contains only the left end of the computed result.

Examples:

```
X := 'abcd'; insert('xy', X, 3); writeln(X)
```

produces the output  
abxycd

```
X := 'tide'; insert('time', X, 10); writeln(X)
```

produces the output  
tidetime.

```
X := 'Gs'; insert('love', X, 2); writeln(X)
```

produces the output  
Gloves.

---

## Summary of Pascal Constructs

### function declaration heading

Syntax:

```
function <function name> (<formal parameter list>) : <type returned>;
```

Example:

```
function Area(Length, Width: real): real;
```

The <formal parameter list> can be anything that is allowed as a procedure formal parameter list. The <type returned> may be the type `real` or any ordinal type (such as `integer`, `char`, or `boolean`) or a TURBO Pascal *string* type. (Most of the types we will introduce later on are not allowed. In fact, the only other types allowed are the pointer types, a class of types not introduced until Chapter 17.)

### function declaration

The syntax for a function declaration is the same as that for a procedure declaration except for two points: The heading of a function declaration is as described above, and the body of the declaration must contain an assignment statement with the function name on the left-hand side of the assignment operator.

### function call

Syntax:

```
<function name> (<argument list>)
```

Example:

```
X := Area(3.79, 8.9)
```

A function call can appear in exactly the same places that a constant of the type returned by the function can appear. It is an expression and evaluates to a value of the type specified in the function declaration. This value is called the value returned. The value returned is equal to the last value assigned to the function name when the statements given in the function declaration are executed. The <argument list> is the same

---



as an actual parameter list and is handled in exactly the same way as an actual parameter list for a procedure. At the time that the function call is evaluated, all the statements in the body of the function declaration are executed, and so any side effects, such as setting a variable parameter, will take place at that time.

### subrange type declarations

Syntax:

```
type <type name 1> = <lower limit 1> . . <upper limit 1>;
    <type name 2> = <lower limit 2> . . <upper limit 2>;
    .
    .
    .
    <type name n> = <lower limit n> . . <upper limit n>;
```

Example:

```
type Index = 0 . . 100;
    Grades = 'A' . . 'F';
```

The type names are identifiers chosen by the programmer. For each  $i$ ,  $\langle \text{lower limit } i \rangle$  and  $\langle \text{upper limit } i \rangle$  must be constants of the same ordinal type.  $\langle \text{lower limit } i \rangle$  must be less than or equal to  $\langle \text{upper limit } i \rangle$ .

### subrange variable declarations

Example:

```
var I: Index;
    Latin, Math, Psych: Grades;
```

Variables of a subrange type are declared just like variables of predefined types such as `integer`. A variable of a subrange type may only take on values in the range specified in the type declaration.

---

## Exercises

### Self-Test Exercises

10. Which of the following are legal type definitions?

```
const Max = 1000;
type SmallNegInteger = -100 . . -1;
    SmallNegNumber = -1 . . -100;
    GradePoint = 0.0 . . 4.0;
    Initial = 'A' . . 'Z';
    Range = 0 . . Max;
    SmallRange = 0 . . Max - 100;
```

---

11. Give suitable type declarations for data of each of the following kinds: exam scores in the range 0 to 100; the nonnegative integers; the integers whose absolute value is at most 100.
12. (This exercise uses the optional section “Enumerated Types.”) Can a program read in a value of an enumerated type from the keyboard? Can it write one to the screen?

### Interactive Exercises

13. Determine the ordering of the letters on your system. Do lowercase letters come before uppercase letters or after them, or are they intermixed? Are there any special characters mixed in with the letters? For example, is there any character between 'A' and 'B'?
14. (This exercise uses the optional section on pseudorandom number generators.) Write a program that outputs 20 pseudorandom integers in the range 5 . . 20.
15. (This exercise uses the optional sections on pseudorandom number generators.) Write a program that outputs 20 pseudorandom real values between 0 and 10.
16. (This exercise uses the optional section “More Standard Functions.”) For positive values of  $x$ , the value  $x^y$  can be computed as

$$\exp(y * \ln(x))$$

The advantage of this formula over the function `Power` in Figure 8.3 is that this formula allows fractional exponents. Write a program that reads in decimal numbers  $x$  and  $y$  and outputs  $x^y$ .

17. (This exercise uses the optional section “The Functions `pred`, `succ`, `ord`, and `chr`.”) Determine the “ring bell” character number for your system. Feel free to look it up in a manual or to ask a friend.

### Programming Exercises

18. Write a function declaration for a function called `Grader` that takes a numeric score and returns a letter grade. `Grader` has one argument of type `integer` and returns a value of type `char`. Use the rule that 90 to 100 is an A, 80 to 89 is a B, 70 to 79 is a C, and less than 70 is an F. Embed it in a test program.
19. Write a function declaration for a function that computes interest on a credit card account balance. The function has arguments for the initial balance, the monthly interest rate, and the number of months for which interest must be paid. The value returned is the interest due. Do not forget to compound the interest, that is, to charge interest on the interest due. The interest due is added into the balance due, and the interest for the next month is computed using this larger balance. Embed the function in a test program.
20. (This exercise uses the optional section “More Standard Functions.”) The perimeter  $P$  of an  $n$ -sided polygon circumscribing a circle of radius  $r$  is given by

$$P = 2nr \times \tan(\pi/n)$$

Write a function declaration that will return the perimeter of such a polygon given the values of  $n$  and  $r$  as arguments. Embed the function in a program as a test of the function declaration. The program should include a test to see if  $n$  is less than 3, since the formula, and hence the function, will not work unless  $n$  is 3 or more.

21. Write a program that gives the user the choice of computing the area of any of the following: a circle, a square, a rectangle, or a triangle. The program should include a loop to allow the user to perform as many calculations as desired. Use a function for each of the different kinds of calculations.

22. In order to discourage excess consumption, an electric company charges its customers a lower rate, namely \$0.11, for each of the first 250 kilowatt hours, and a higher rate of \$0.17 for each additional kilowatt hour. In addition, a 10% surtax is added to the final bill. Write a program to calculate electric bills given the number of kilowatt hours consumed as input. Use two function declarations: one to compute the amount due without the surtax and one to compute the total due. The declaration for the second function should include a call to the first function.

23. (This exercise uses the optional case study “Testing for Primes.”) The function `Prime` in Figure 8.7 can be made more efficient in a number of ways. First, there is no reason to continue to loop through more checks once a divisor of  $N$  is found. At that point you know that  $N$  is not prime. For example, once you know that 1000 is divisible by 2 you know it is not a prime and need not test to see if it is divisible by 3, 4, and so forth. Replace the *for* loop with a *repeat* or *while* loop that will terminate the loop as soon as it is discovered that an argument is not a prime. Also, there is no need to test all numbers up to  $N-1$ . Determine and use a smaller limit on the maximum number of loop iterations.

24. The greatest common divisor of two positive integers is the largest integer that divides them both. For example, the greatest common divisor of 9 and 6 is 3. Write a function declaration for a function with two integer arguments that returns their greatest common divisor.

25. (This exercise uses the optional sections on pseudorandom number generators.) Write a program that outputs random but grammatically correct sentences. The sentences can be simple sentences that take the following form: a noun followed by a verb, followed by a noun. Use a pseudorandom number generator to choose nouns from a list of 10 nouns and verbs from a list of 10 verbs.

26. (This exercise uses the optional sections on pseudorandom number generators.) Write a program that will simulate a roll of two dice to produce a value between 2 and 12. If your system has a predefined pseudorandom number generator, use it; otherwise, use the function `Random` in Figure 8.9.

27. The game of Nim is played as follows. There are three piles of sticks. Two players take turns making moves. A move consists of picking up as many sticks as the player desires, subject to the following constraints: All the sticks must be picked up from the same pile, and a player must pick up at least one stick. The player who picks up the last stick loses. Write a program to play Nim with the user. The piles of sticks should initially contain three, five, and seven sticks each. The computer can use any strategy you



wish so long as it obeys the rules. Display the piles as three lists of a symbol of your choice, such as '!', so that the piles are displayed as something like

```
!!!  
!!!!!  
!!!!!!!
```

28. (This exercise uses the optional sections on pseudorandom number generators.) Redo the previous exercise (or do it for the first time), but this time use a pseudorandom number generator to choose the size of the three piles, subject to the constraints that the three piles must be of different sizes and must contain at least 2 and at most 10 sticks. The computer should use a pseudorandom number generator to decide on moves.

29. (This exercise uses the optional sections on pseudorandom number generators.) Write a function declaration for a function called `Deal` that returns a card chosen at random from a standard deck of cards. Code the value of the card as an integer value: 2 through 10 as itself, jack as 11, queen as 12, king as 13, and ace as 14. Your program need not remember what cards are dealt, and so, for example, it might deal five aces. You can ignore suits (diamonds, clubs, etc.).

30. (This exercise uses the optional sections on pseudorandom number generators.) Write a program to play blackjack that uses the procedure `Deal` from the previous exercise. The program deals two cards to the user and two cards to itself (all cards are shown to the user). The user can then request more cards until he or she is busted or wants to stop. The program then deals additional cards to itself until its score exceeds 16. Allow the user to play additional hands until he or she wants to stop. The program in Figure 6.12 may be of some help.

31. (This exercise uses the optional sections on pseudorandom number generators.) One way to estimate the area of a figure is to enclose it in a figure whose area you know and to then choose points at random within the figure of known area. The ratio of the number of points in the figure of unknown area to the number of points in the enclosing figure of known area is the same as the ratio of the unknown area to the known area. (Remember to count the total number of points as the points in the figure of known area. Those that are in the figure of unknown area are also in the larger figure.) Use this technique with a circle enclosed in a square to estimate the area of a circle with a radius of 1. Then use the formula

$$Area = \pi r^2$$

with that estimate of *Area* and solve for  $\pi$  to get an estimate of  $\pi$ . Do this all with a program that uses a pseudorandom number generator to choose points.

32. Write a program to compute the monthly cost of a house based on the following four input values: the purchase price, the annual fuel costs, the tax rate expresses as tax per \$1000, and the down payment. The monthly cost is 1/12th of the annual cost obtained by summing the costs of tax, fuel, and mortgage. The annual tax is obtained by multiplying the tax rate by the purchase price. For example if the rate is \$20.50 per \$1000 and the purchase price is \$100,000, then the annual tax is \$2,050. The annual

mortgage cost is 10% of the balance left after deducting the down payment from the purchase price. The program should repeat the calculation for new inputs as often as the user desires.

33. (This exercise uses the optional section "More Standard Functions.") Write a function declaration for a function called `SineDeg`. This function differs from the standard function `sin` in that its argument is in degrees rather than radians. Use a local function that converts from degrees to radians, and use a call to `sin`. Write similar function declarations for `cos` and `tan`. Embed these in a program that takes as input an angular measure in degrees and then outputs the sine, cosine, and tangent of that angle.

34. (This exercise is for TURBO Pascal users.) Write a program that reads in a sentence containing your name and then echoes it with your name replaced by your nickname and your nickname replaced with your full name. For example, if your name is "Robert Smith" and your nickname is "Bob," then the input sentence "Robert Smith is often called Bob." should produce the output sentence "Bob is often called Robert Smith." The names will be part of your program so that the program works only for your names. However, to make it work for somebody else's names, you should only need to change the constant declarations.

35. (This exercise is for TURBO Pascal users.) Write a program that reads the user's first, middle, and last names in that order and then echoes the names sorted alphabetically. For example, the input names

**Walter John Savitch**

should produce the output names

John Savitch Walter

36. (This exercise is for TURBO Pascal users.) Write a program that will read in a sentence of up to 50 characters and then output the sentence with letters corrected for capitalization. In other words, the output sentence should start with an uppercase letter but should contain no other uppercase letters. Do not worry about proper names. For example, the input

**the Answer IS 42.**

should produce the output

The answer is 42.

37. (This exercise is for TURBO Pascal users.) Enhance the program in Figure 8.16 so that it also replaces the masculine endings "-man" and "-men" with "-person" and "-people," respectively; for example, "mailman" would be changed to "mailperson," and "gentlemen" would be changed to "gentlepeople." After that, enhance it further so that it replaces the phrase "ladies and gentlemen" with the word "people." Be sure that "ladies and gentlemen" is not mistakenly changed to "ladies and gentlepeople." Your program should handle capitalization in the normal way so that "Ladies and gentlemen . . ." is changed to "People . . ."; this applies to all the substitutions, not just to the word "Ladies."

38. (This exercise is for TURBO Pascal users.) Write a program that will read in a one-line yes/no question and will then answer the question with a complete sentence. For example, the question "Do you love me?" should produce one of the two responses "Yes, I do love you." or "No, I do not love you." Use the predefined TURBO function random to choose between yes and no answers. To simplify the problem, assume that the subject of the sentence is of the form "the" or "a" or "all" followed by a noun or else is just a single noun without any articles or adjectives. Assume, too, that the verb, like "Do," is the first word of the question and that the one- or two-word subject immediately follows this verb. For example, in "Will all men die someday?", the verb is "Will" and the subject is "all men." (What we are calling *the verb* will often turn out to be what is usually called *the auxiliary verb*.) To get an affirmative answer, your program interchanges this initial verb and the subject. It then changes "you" to "I," "I" to "you," "me" to "you," etc. Then it adds "Yes," fixes up capitalization, and changes the question mark to a period. This is long, but it is not as hard as it looks. There is a difficulty with "you" since it is not easy to tell whether it should be changed to "I" or "me." You may simply always change "you" to "I," even though this change will sometimes produce an ungrammatical sentence. It is acceptable to generate some rather awkward answers like "No, all men will not die someday." For extra credit enhance your program to produce less awkward answers.

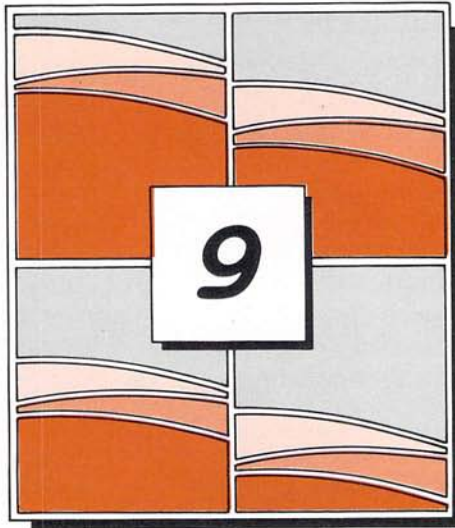
---

## References for Further Reading

D.E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, second edition, 1981, Addison-Wesley, Reading, Mass. More-advanced material on random number generators. Does not use Pascal, but the text can be read without reading the programs.

---





## *Arrays for Problem Solving*

It is a capital mistake  
to theorize before one has data.  
*Sir Arthur Conan Doyle (Sherlock Holmes),  
Scandal in Bohemia*

## Chapter Contents

Introduction to Arrays	Array Example with Noninteger Indexes
Pitfall—Use of Plurals in Array Definitions	Random versus Sequential Array Access
Type Declarations—A Summary	Array Indexes with Semantic Content
Input and Output with Arrays	Enumerated Types as Array Index Types (Optional)
Partially Filled Arrays	Case Study—Production Graph Off-Line Data (Optional)
Pitfall—Array Index Out of Range	Case Study—Sorting
TURBO Pascal—More about the R Compiler Directive	Summary of Problem Solving and Programming Techniques
Self-Test Exercises	Summary of Pascal Constructs
The Notion of a Data Type	Exercises
Arrays as a Structured Type	References for Further Reading
Allowable Function Types	
Case Study—Searching an Array	
Pitfall—Type Mismatches with Array Parameters	
Efficiency of Variable Parameters	

**I**n this chapter we introduce a common and extremely useful class of defined data types known as *arrays*. These will be our first examples of *structured types* and will serve to introduce both the idea of a structured type and the importance of these types to problem solving. A structured type can be described briefly as a complex type built up from simpler types. A more detailed discussion of the concept is included in the chapter.

## Introduction to Arrays

Suppose we wish to write a program that reads in five test scores and then manipulates the scores in some way. For instance, the program might compute the highest test score and then output the amount by which each score falls short of the highest score. The highest score is not known until all five scores are read in. Hence, all five scores must be retained in storage so that after the highest score is computed each individual score can be compared to it. To retain the five scores, we will need something equivalent to five variables of type `integer`. We could use five individual variables of type `integer`, but five variables are hard to keep track of, so this is not an elegant solution. We could make the program more readable by giving the variables related names, such as `Score1`, `Score2`, and so forth, but this solution becomes absurd if the number of scores is very large. Imagine doing this if there were 100 scores instead of just 5. The solution we will propose is similar to this idea of using a list of variables, but it will handle the details in a much neater fashion.

To solve this dilemma, we introduce a new Pascal construct known as an *array*. An array is rather like a list of variables, each of which has a two-part name. One part of the name is the same for each of the variables that collectively constitute the array. The other part is different for each variable. For example, the five names for the five individual variables we need might be `Score[1]`, `Score[2]`, `Score[3]`, `Score[4]`, and `Score[5]`. The part that does not change, in this case `Score`, is the name of the array. In this example, the part that can change is an integer in the subrange `1 .. 5`.

In Pascal the type and variable declarations for an array of the kind just described can be given as follows:

```
type SmallArray = array[1 .. 5] of integer;
var Score: SmallArray;
```

The type given after the word *of*, such as `integer` in the above declaration, is called the *component type* or *base type*. This declaration is like declaring the following five variables to all be of type `integer`:

*component  
type*

```
Score[1], Score[2], Score[3], Score[4], Score[5]
```

Variables like the above five that are derived from an array name are called *indexed variables* in order to distinguish them from the sort of variables we have seen up to now. These five indexed variable names are not valid Pascal identifiers since they contain the square bracket symbols `[ ]`, and so they may not look like variables. However, they have all the properties of variables. An indexed variable like `Score[1]` can be used *anyplace* that an ordinary variable of type `integer` can be used. For example, with `Score` declared in this way all of the following are possible:

*indexed  
variables*

```
readln(Score[1], X, Score[5]);
writeln(Score[1], Score[2], X);
X := Score[3];
Score[4] := X;
{X is a variable of type integer.}
```



In addition to having all the properties of simple variables, these indexed variables have other properties that simple variables do not have. The most important such property is that a variable or more complicated expression can be used inside the square brackets.

*index*

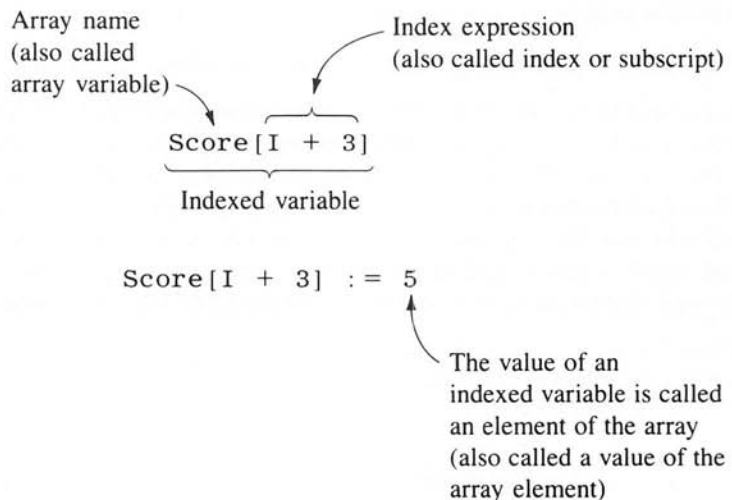
The expression inside the square brackets is called the *index expression* or, more simply, the *index*. These indexes have some similarity to the subscripts on the subscripted variables used in mathematics, like the  $i$  in  $x_i$ , and hence some programmers use the term *subscript* as a synonym for “index.” Array indexes are truly integer values. For our sample array `Score`, any integer expression that evaluates to a value of the subrange type `1 .. 5` can be placed inside the square brackets. This provides the program with a way of manipulating the names of the indexed variables. For example, the following code sets the value of `Score[2]` to 99 and outputs it to the screen twice:

```
X := 2;
Score[X] := 99;
writeln(Score[X]);
writeln(Score[2])
```

The variable `X` may be of type `1 .. 5` or of type `integer`. Figure 9.1 presents a summary of the terms used in discussing indexed variables.

*elements*

Two things should be observed in the preceding piece of code. First, the array index can be a variable. This allows the program to say things equivalent to “do the following to the  $X$ th indexed variable.” Second, the identity of an indexed variable, such as `Score[X]`, is determined by the value of its index (and, of course, by the array name, like `Score`). In the example of the previous paragraph, `Score[2]` and `Score[X]` are the exact same indexed variable, because the value of `X` is 2. Similarly, if the value of `X` is 2, the indexed variable `Score[X + 3]` is the exact same variable as `Score[5]`.



**Figure 9.1**  
Indexed variable.

Score [1]	Score [2]	Score [3]	Score [4]	Score [5]
9	6	2	10	7

Values of array elements before code is executed

### Sample Code

```
writeln(Score[2]); {Display the value of Score[2] = 6 on the screen.}
Score[2] := 9;      {Change the value of Score[2] from 6 to 9.}
I := 2;
Score[I + 3] := 8; {Change the value of Score[5] from 7 to 8.}
Score[1] := Score[4] - 6; {Change the value of Score[1] from 9 to
                           10 - 6 = 4.}
```

Score [1]	Score [2]	Score [3]	Score [4]	Score [5]
4	9	2	10	8

Values of array elements after code is executed

**Figure 9.2**  
**Manipulating**  
**array elements.**

Figure 9.2 provides an explanation of how a sample piece of Pascal code manipulates the indexes and elements of the array `Score`.

For another example, suppose that the value of `X` is 2 and that `Pro` is a procedure with one variable parameter of type `integer`. Then the procedure call

`Pro (Score [X])`

is equivalent to

`Pro (Score [2])`

Hence, when an indexed variable serves as an actual variable parameter, the index expression is always evaluated before the indexed variable is substituted for the formal parameter.

An array type is declared in the same place and in the same general manner as a subrange type. As with any type declaration, the declaration for an array type consists of the type name followed by an equal sign, the type definition, and a semicolon. The form of an array type definition is given by the syntax diagram in Figure 9.3. For example, consider the declaration for the array `Score` given earlier in this section. The identifier `SmallArray` is declared to be an array type. In that declaration we used `1 . . 5` as a <subrange type definition>. Any subrange type definition or any previously declared name of a subrange type can be used instead. As the component type we used the type `integer`. Any Pascal type, including the types we will introduce in later chapters, can be used as the component type.

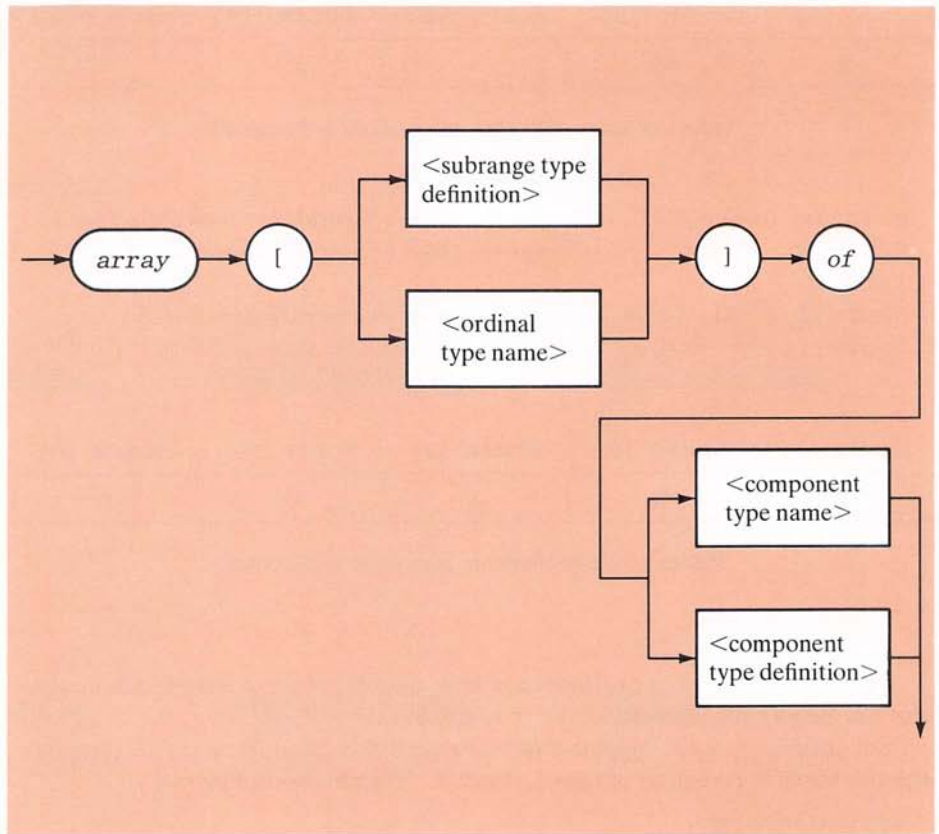
The syntax diagram in Figure 9.3 indicates that you can use any ordinal type name to specify the index type of an array. Hence, you can use the types `boolean` and `char` as index types. However, it is not practical to use the full type `integer`

*indexed  
variables as  
parameters*

*type  
declaration*

*component  
type*

*index  
type*



**Figure 9.3**  
Syntax diagram for  
an array type  
definition.

as the index type of an array, even though it is an ordinal type. Definitions of the form `array [integer] of . . .` will *not* work because the index type is too large.

Whatever the index type is, an array with that index type has one indexed variable for each value of the index type. For example, consider the following declarations:

```

type Name1 = array[boolean] of integer;
      Name2 = array[0 .. 2] of char;
      Name3 = array['a' .. 'c'] of real;
var A: Name1;
    B: Name2;
    C: Name3;
  
```

The array A has two indexed variables: A[false] and A[true], each capable of holding one integer. The array B has three indexed variables: B[0], B[1], and B[2], each capable of holding one character value. The array C also has three indexed variables: C['a'], C['b'], and C['c'], each capable of holding one value of type real.



**Program**

```

program ShowScores(input, output);
{Reads in 5 scores and shows how much each score differs from the highest score.}
type SmallArray = array[1..5] of integer;
var Score: SmallArray;
    I, Max: integer;
begin{Program}
  writeln('Enter five scores:');
  read(Score[1]);
  Max := Score[1]; {Largest so far.}
  for I := 2 to 5 do
    begin{for}
      read(Score[I]);
      if Score[I] > Max then
        Max := Score[I]
      {Max is the largest of the values of Score[I], . . . , Score[I].}
    end; {for}
  writeln('The highest is ', Max);
  writeln('The scores and');
  writeln('their difference from the highest are:');
  for I := 1 to 5 do
    writeln(Score[I], ' off by ', Max - Score[I])
end. {Program}

```

**Sample Dialogue**

```

Enter five scores:
5 9 2 10 6
The highest is    10
The scores and
their differences from the highest are:
  5 off by  5
  9 off by  1
  2 off by  8
 10 off by  0
  6 off by  4

```

**Figure 9.4**  
**Program using an**  
**array.**

Notice that once an array type name has been declared, a particular array is declared just like a variable. Neither the square brackets nor the index expression is included in the declaration of the array.

A complete program that uses an array to display scores in the manner described in the opening discussion of this section can be found in Figure 9.4. The algorithm for finding the highest score is essentially the same as that used for the program in Figure 7.9.

*array*  
*declaration*

## Pitfall

### Use of Plurals in Array Definitions

An array of reals contains more than one number. Hence, the following type declaration seems natural:

```
type List = array[1 . . 25] of reals;
```

As innocent as it may look, it will produce an error message. There is no type named reals. The “s” that seems so natural to speakers of English is incomprehensible to the compiler. If you delete the “s,” the declaration is correct. In array type definitions, the component type name is used unchanged, even though that may violate your sense of English grammar.

## Type Declarations—A Summary

All type definitions are given together. Types may be defined in terms of named constants and in terms of other types. In fact, doing so can aid readability. For example, a program might open as follows:

```
program Sample(input, output);
const Start = 0;
      Stop = 100;
type Subscript = Start . . Stop;
      List = array[Subscript] of real;
var X, Y: real;
    I: Subscript;
    A, B: List;
```

Notice that no type name is used before it is defined.

*local  
types*

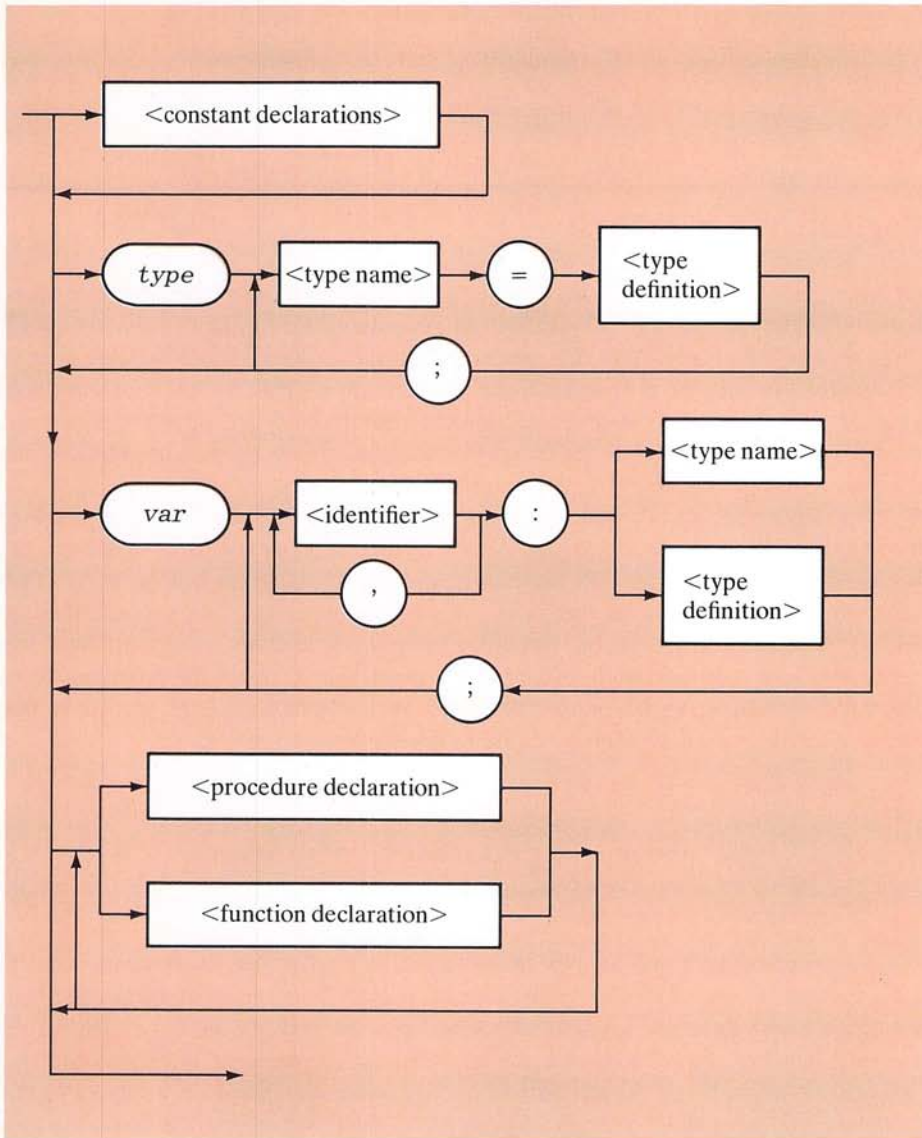
As with the other kinds of declarations, type declarations may be local to a procedure, although the need for local type declarations is rare. Like constants, types are usually best declared globally. The syntax for the declaration section of a program or procedure is summarized in Figure 9.5.

It is possible to use a type definition directly instead of first defining a type name. Hence, the following is a legal program opening:

```
program Sample(input, output);
var A, B: array[1 . . 100] of real;
```

*parameter  
types*

However, it is better to declare names for defined types and to refer to the types by name. When you are specifying a type for a procedure or function parameter, it is absolutely necessary to use a type name. A formal parameter list cannot contain any type definitions, only type names.



**Figure 9.5**  
Declaration section  
of a block.

## Input and Output with Arrays

You cannot use an entire array as a parameter to a `read` or `readln` statement. For instance, if the array `Score` is declared as in Figure 9.4, then the following is not allowed:

```
read (Score) {NOT ALLOWED if Score is an array type.}
```



Similarly, it is not possible to output an array with a statement like

```
write (Score) {NOT ALLOWED if Score is an array type.}
```

Typically, input and output of arrays is done with *for* loops, as in Figure 9.4.

---

## Partially Filled Arrays

Often the exact size needed for an array is not known at the time a program is written. Sometimes the required size will differ from one run of the program to another. Some programming languages allow the size of an array to be determined by some sort of input at the time that the program is run; Pascal does not. Thus, if we do not know how large an array is needed, we must declare the array to be of the largest size that the program could possibly need. The program is then free to use as much or as little of the array as it needs.

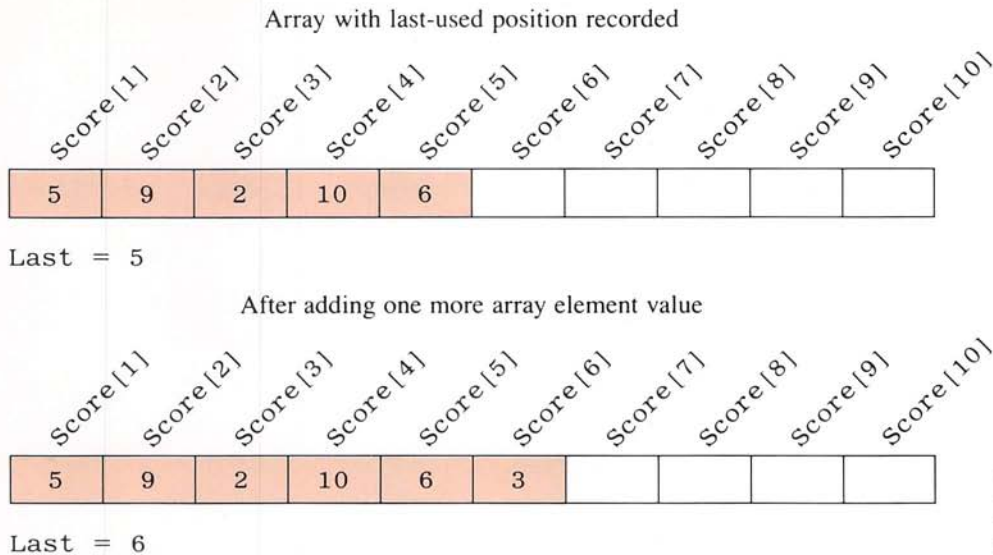
Partially filled arrays require some special care. The program must keep track of how much of the array is used and must not reference any indexed variable that has not been given a value. To illustrate the point, consider the program in Figure 9.4. That program works fine provided that there are exactly five scores, but suppose that the number of scores varies from one run of the program to another. Perhaps different students who have taken different numbers of exams will be using the program. If we know the maximum possible number of scores, then we can declare the array to be that large. If the maximum number is 10, the following declarations will do:

```
const MaxIndex = 10;  
type Index = 1 . . MaxIndex;  
      SmallArray = array[Index] of integer;  
var Score: SmallArray;  
      Last, I: Index;
```

Since there may be less than the maximum number of scores, the program needs to keep track of which indexed variables have been used. This can be accomplished with a variable *Last* to record the last index used, as illustrated in Figure 9.6. The scores are read into the array *Score*, and every time another number is entered into the array, the value of *Last* is increased by one. A complete program that illustrates this technique is displayed in Figure 9.7. Notice that when writing out the array, the program only outputs those components that have a value.

*empty  
arrays*

The sample program in Figure 9.7 had no need to allow for an empty list of scores. However, in other situations a program may have to cope with an empty list of data items. For example, a list of sales figures might be empty if no sales were made. In these cases, a program may need to record the fact that no array elements have been used. If the index type is 1 . . *MaxIndex* and the variable *Last* records the last index used, then an empty list can be indicated by setting *Last* equal to 0. If this is done, then the type of *Last* cannot be the index type of the array, since *Last* may assume a value outside that range. In this situation *Last* would have to be declared to be of type *integer* or of type 0 . . *MaxIndex*.



**Figure 9.6**  
Partially filled  
array.

Although we cannot write our programs so that the size of an array can be determined by the user, we can write our programs so that the array sizes can be increased or decreased by making only minor changes in the program. If we declare one or both bounds of the index type to be defined constants, like `MaxIndex` in our sample, we can then write the program in such a way that simply changing these defined constants will produce a correct program for different-sized arrays.

*providing  
for array  
expansion*

## Pitfall

### Array Index Out of Range

The most common programming error that is made with arrays is attempting to reference a nonexistent index. If the index type is `1 . . MaxIndex` for the array `Score`, then a reference to `Score[I]` when the value of `I` is either less than 1 or greater than `MaxIndex` is meaningless and should precipitate an error message.

When input is read into an array with a loop, one way to guard against referencing a nonexistent array index is to test to see whether the array is full and to terminate the loop if it is. In Figure 9.8 we have rewritten the loop from the program in Figure 9.7 so that it does just that. Be sure to notice that we have inserted code to output a warning if the array is too small. Without such a warning message, we would avoid an error message but would likely produce an even worse situation: an undetected error. If we omit the warning message and there is too much input for the array, then the user might not notice that something is wrong.

*exceeding  
array  
capacity*

**Program**

```

program ShowScores2(input, output);
{Reads up to 10 scores and shows how much each score differs from
the highest score. Assumes there is at least one score.}
const MaxIndex = 10;
type Index = 1 .. MaxIndex;
   SmallArray = array[Index] of integer;
var Score: SmallArray;
    Last, I: Index;
    Max: integer;
begin{Program}
  writeln('Enter up to ', MaxIndex, ' scores');
  writeln('and then press return: ');
  read(Score[1]);
  Max := Score[1]; {Largest so far.}
  Last := 1;
  while not eoln do
    begin{while}
      Last := Last + 1;
      read(Score[Last]);
      if Score[Last] > Max then
        Max := Score[Last]
      {Max is the largest of the values of Score[1], . . . ,Score[Last].}
    end; {while}
  writeln('The highest is ', Max);
  writeln('The scores and');
  writeln('their difference from the highest are:');
  for I := 1 to Last do
    writeln(Score[I], ' off by ', Max - Score[I])
end. {Program}

```

**Sample Dialogue**

```

Enter up to      10 scores
and then press return:
5 9 2 10 6 3
The highest is   10
The scores and
their differences from the highest are:
  5 off by  5
  9 off by  1
  2 off by  8
 10 off by  0
  6 off by  4
  3 off by  7

```

**Figure 9.7**  
**Program using a**  
**partially filled**  
**array.**



```

{Last must be initialized before here.}
while (not eoln) and (Last < MaxIndex) do
  begin{while}
    Last := Last + 1;
    read(Score[Last]);
    if Score[Last] > Max then
      Max := Score[Last]
      {Max is the largest of the values of Score[1], . . . ,Score[Last].}
    end; {while}
  if not eoln {and hence Last >= MaxIndex} then
    writeln('Warning: Some numbers were not read in. ');

```

**Figure 9.8**  
Checking for  
exceeding array  
capacity.

## TURBO Pascal

### More about the R Compiler Directive

TURBO Pascal will *not* always give an error message if your program attempts to reference a nonexistent array index. However, you can ask it to give such error messages by inserting the following line in the file that contains your program:

```
{R+}
```

This is the same compiler directive that we discussed in Chapter 8. This single compiler directive turns on both subrange error messages, as explained in Chapter 8, and array index error messages. As noted in Chapter 8, this line should come before your program and should not contain any blanks. The “R” stands for “Range checking,” since it instructs the computer to check for index values that are “out of Range.”

It is a good idea to include this R compiler directive while debugging your program. After your program is completely debugged, you may wish to remove this directive. Your program may run somewhat faster if the directive is omitted.

A comment can be added to the directive as follows:

```
{R+ turns on array index error messages.}
```

In fact, the entire line, including the directive, is a comment, but it is still read by the compiler. It is the dollar sign that tells the compiler to read the comment. The R+ tells the compiler to turn on array index error messages. The explanatory comment after the R+ directive is ignored by the compiler. There should be *no blanks* between the opening curly bracket and the plus sign, although blanks are allowed after that point.

We will introduce other compiler directives as we need them. Common compiler directives are summarized in Appendix 16.

## Self-Test Exercises

1. Which of the following are legal type definitions?

```

type AnsList = array[0 .. 10] of boolean;
   Index = -100 .. -50;
   List = array[Index] of real;
   Count = array[char] of integer;
   NonNeg = 0 .. maxint;
   Tally = array['a' .. 'z'] of NonNeg;
   AnswerCount = array[boolean] of NonNeg;
   TempCount = array[real] of integer;
   GradeTally = array[0.0 .. 4.0] of integer;

```

2. Write suitable type declarations for each of the following:
- An array to hold 100 scores, each between 0 and 10.
  - An array of reals indexed by the type 'a' .. 'z'.
  - An array of characters whose smallest index is -5 and whose largest index is 19.
3. Give suitable type and variable declarations for data of each of the following kinds:
- A list of 100 or fewer scores, each a whole number in the range 0 to 10.
  - An array to record for each letter of the alphabet the number of students in a class whose last name starts with that letter.
  - An array to record which students have completed graduation requirements. The students are numbered 1 through 100. The array records whether or not they can graduate and nothing else.
4. The following piece of code is supposed to add all the elements in an array A, which has 100 elements. It does not work. What is wrong with it, and how should it be fixed?

```

Sum := 0;
for Element := A[1] to A[100] do
    Sum := Sum + Element

```

5. Write code to initialize an array C, declared as follows, so that each element has a value of zero.

```

type RealList = array[0 .. 100] of real;
var C: RealList;

```

6. If C is declared as in the previous exercise, is the following legal?

```

C := 0

```

7. The following piece of code is supposed to test an array of elements to see whether they are in order. It contains a bug. What is it?

```

var InOrder: boolean;
    I: integer;
    A: array[First .. Last] of integer;
    . . .
    . . .
InOrder := true;
for I := First to Last do
    if A[I] > A[I + 1] then
        InOrder := false

```

8. Write code that reads exactly six letters into an array and then outputs them in reverse order. The array is declared as follows:

```

type Word = array[1 .. 6] of char;
var A: Word;

```

9. Write code that reads a list of up to 10 positive integers into the array A, declared as shown below, and then writes the integers back to the screen. The input list is terminated with either a negative number or zero. The terminating number is not written back out.

```

type PositiveInt = 1 .. maxint;
    List = array[1 .. 10] of PositiveInt;
var A: List;

```

---

“But it’s always interesting when one doesn’t see,” she added.  
 “If you don’t see what a thing means, you must be looking  
 at it wrong way round.”

*Agatha Christie*

---



---

## The Notion of a Data Type

The word “data” in its most general sense refers to anything that can be manipulated by a computer program. A *data type* is a particular type or kind of data together with some rules for how these data items can be manipulated. The data types *integer*, *real*, *char*, and *boolean* are provided automatically in the Pascal language. Additionally, we have seen how subrange types and array types can be defined within a Pascal program.

One way to think of a data type is as a description of the values that a variable of that type can have. A data type is specified by specifying the values of that type and by specifying the operations that are allowed on those values. For example, the values of the Pascal type *integer* are all the positive integers that are less than or equal

---



to `maxint`, all the negative integers that are greater than or equal to the smallest negative integer the computer can handle (approximately `-maxint`), and the integer zero. The operations that are allowed include addition, subtraction, multiplication, *div*, *mod*, and the comparison operators, such as `=` and `<`. The subrange type `1..5` has the same operations but a different set of values, namely 1, 2, 3, 4, and 5. The type `boolean` consists of the two values `true` and `false`, and the operations allowed consist of *and*, *or*, *not*, and a few comparison operations, such as `=` and `<>`.

The discussion in the previous paragraph is easy to apply to simple data types such as `integer` and `boolean`. However, at first reading it may not be apparent that it also applies to array types. In the next section, we will see that, when viewed properly, it does.

---

## Arrays as a Structured Type

One can consider an array to be a collection of (indexed) variables that are named in a convenient and uniform way. For instance, the array `A`, declared as follows, can be thought of as five indexed variables, `A[0]`, `A[1]`, and so forth, each capable of holding one value of type `real`.

```
type RealList = array[0..4] of real;
var A, B: RealList;
```

This point of view is often adequate. However, to understand the complete nature of arrays, you must also take another view.

An array can also be viewed as a single variable of a complex type. Our array `A` can be thought of as a single variable whose value is a list of five `real` numbers, for example

```
2.3, 4.0, 3.6, 2.8, 3.6
```

To emphasize this point of view, variables like `A` are called *array variables*. Do not confuse the terms “array variable” and “indexed variable:” `A` is an array variable; `A[3]` is an indexed variable. Figure 9.9 illustrates the terminology and presents one way of visualizing arrays and array values.

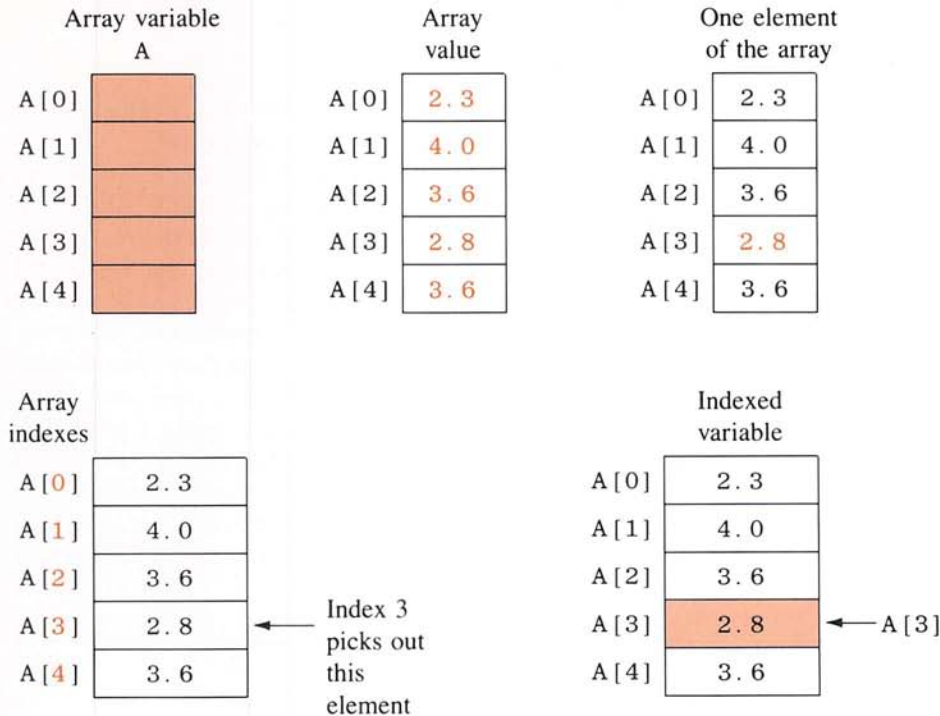
Since an array variable can be viewed as a single variable with a single (compound) value, you can often treat it just like any other variable. For example, with the array variables `A` and `B` declared as above, the following assignment statement is perfectly valid:

```
B := A
```

This statement sets the value of `B` equal to the value of `A`. After this statement is executed, the value of `B[0]` will be the same as that of `A[0]`, the value of `B[1]` will be the same as that of `A[1]`, and so forth for each component of the array `B`.

We have already noted that an indexed variable, such as `A[1]`, can be used as a parameter in a procedure call. It is also possible to use an entire array, such as `A`, as a single parameter to a procedure or function. In such cases, the array is treated as a

*array  
parameters*



**Figure 9.9**  
Array and  
array value.

single variable with a single (compound) value. In this situation, the formal parameter in the procedure heading and the actual parameter in the procedure call are specified by giving the array name without square brackets or indexes. For example, if the array type `RealList` is declared as above, then the following is a legitimate procedure heading:

```
procedure Sample(X: RealList; var Y: RealList);
```

If `A` and `B` are declared to be of type `RealList`, the following is a legitimate procedure call:

```
Sample(A, B)
```

A complete example of the use of array parameters is given in the next case study.

Be sure to note the difference between using an array variable as a parameter and using an indexed variable as a parameter. To illustrate this difference, let us continue to assume that `A` is an array of type `RealList`, as in the previous discussion. When the entire array is passed as a parameter, then the parameter type must be `RealList` and the actual parameter is `A`, written without any square brackets or indexes. When only one indexed variable is passed, such as `A[3]`, then the parameter type must be `real` and both square brackets and an index are used when giving the actual parameter. For example, consider the following procedure heading:

```
procedure SecondSample(X: real; var Y: real);
```

A possible procedure call is

```
SecondSample (A[3], A[1])
```

In this case, we want only indexed variables, not entire arrays, as actual parameters. In the previous case, we wanted entire arrays as actual parameters.

### *structured types*

Array types are a good illustration of the notion of a *structured type*. All the types introduced in previous chapters are simple types. Simple types, whether provided by Pascal or defined by the programmer, have values that intuitively are indivisible units. The character 'A' cannot be meaningfully decomposed into parts. The real number 2.34 could intuitively be decomposed in a few different ways, but we usually think of it as a single item, and the Pascal language treats it as such. The same holds true for the other simple types. In addition to these simple types, Pascal and most other programming languages allow the programmer to define more-complicated types whose values are compound items composed of a number of values of some simpler type or types. These sorts of compound types are called "structured types" because, unlike the simple types, they have a structure that can be meaningfully decomposed by operations provided within the programming language. For example, to reference one element, as opposed to an entire array, we can combine an index and an array variable to obtain an indexed variable, such as A[2].

---

## Allowable Function Types

Viewing an array as a single value naturally leads to the conclusion that a function can return an array as a value. This conclusion is, however, wrong for the Pascal language. In Pascal the value returned by a function cannot be an array type. It may be any simple type, and, with one exception, which we will see in Chapter 17, these are the only possible types for the value returned by a standard Pascal function. (TURBO Pascal also allows functions that return a *string* type.) There is no compelling conceptual reason for this restriction. Its purpose is solely to make the compiler's job easier.

---

## Case Study

---

### Searching an Array

As an example of the use of array parameters, we will construct an algorithm to search a partially filled array for a particular value and we will then implement the algorithm as a Pascal function with an array parameter.

### Problem Definition

The array type is as follows:

```
type Index = 1 .. High;  
IntegerArray = array[Index] of integer;
```



where `High` is an integer constant. Our function will be given three pieces of data: an array `A` of type `IntegerArray`, an integer value `Last`, and one integer `N`. The function is supposed to tell us whether or not the integer `N` is one of the elements

`A[1], A[2], ..., A[Last]`

For example, the list of numbers might be a list of invalid credit card numbers, and so the function could be used to determine whether a given credit card should or should not be accepted by a merchant.

When searching an array, we are likely to want to know both whether the value is in the list and, if it is, where it is in the list. For example, if we are searching a list of missing credit card numbers, then the array index may serve as a record number. Another array indexed by these same numbers may hold telephone numbers to use in reporting that a credit card has been found. To accommodate such usage, we will design our function to return the array index where the number was found. In other words, if `N` is the number being sought and `A[I] = N`, then our function will return `I`. If the number is not in the array, then our function will return zero.

## Discussion

To accomplish the above task, our algorithm will simply try all possible values for the index until either the sought after number is found or all values have been checked. The strategy of trying all possibilities is called the *brute force* approach. It is not always efficient, but it is straightforward and effective.

*brute  
force  
method*

If the possibilities are stored in an array, then the natural way to try all possibilities is to proceed through the array serially from the first, to the second, and so forth until either the number is found or the last value in the array is reached. This *serial search* algorithm proceeds as follows:

*ALGORITHM  
serial search*

```
I := 1;
```

Perform the following until either the number  
is found or the end of the array is reached:

```
begin{loop}
```

```
    if N = A[I] then
        the number is found
```

```
    else
```

```
        I := I + 1
```

```
end; {loop}
```

```
if the number was found then
```

```
    return I
```

```
else
```

```
    return 0
```

This pseudocode can be implemented using a boolean variable called `Found`. `Found` is initialized to `false`, and is changed to `true`, if and when the sought after number is found.

*boolean  
flag*

The complete function embedded in a demonstration program is shown in Fig-

ure 9.10. Notice that the variable `Last` is declared to be of type `ExtendedIndex` to allow it to be initialized to 0. If we had declared it to be of type `Index`, that would have precipitated an error message when `Last` was set equal to 0. This declaration also allows us to represent the empty list with a value of zero. By using `ExtendedIndex` as the type of the function parameter, we can accommodate an empty list. If you check the code, you will see that, for an empty list, the function correctly returns the value 0.

*array parameters*

In the program in Figure 9.10, the entire array `List` is passed as the actual parameter in the function call

`Search(Number, List, Last)`

(continued, page 352)

### Program

```

program ArraySearch(input, output);
{Searches a partially filled array to see if a value is present.}
const High = 10;
type Index = 1 .. High;
      ExtendedIndex = 0 .. High;
      IntegerArray = array[Index] of integer;
var Number: integer;
    List: IntegerArray;
    Last, I: ExtendedIndex;

function Search(N: integer; A: IntegerArray;
               Last: ExtendedIndex): ExtendedIndex;
{Returns I such that N = A[I], provided there is such an I;
 otherwise it returns 0. Last is the last array index used.}
var I: integer;
    Found: boolean;
begin{Search}
    Found := false; {not found so far.}
    I := 1;
    while (I <= Last) and (not Found) do
        if N = A[I] then
            Found := true
        else
            I := I + 1;

    if Found then
        Search := I
    else
        Search := 0
    end; {Search}

```

**Figure 9.10**  
Searching a  
partially filled  
array.

```
begin{Program}
  writeln('Enter a list of at most');
  writeln(High, ' integers,');
  writeln('and then press return:');
  Last := 0;
  while not eoln do
    begin{while}
      Last := Last + 1;
      read(List[Last])
    end; {while}
  readln;
  writeln('Enter a number to be searched for:');
  readln(Number);
  I := Search(Number, List, Last);
  if I > 0 then
    writeln(Number, ' found in position ', I)
  else
    writeln(Number, ' is not on the list.')
end. {Program}
```

#### Sample Dialogue 1

Enter a list of at most  
10 integers,  
and then press return:  
10 9 8 7 6 5 4 3 2 1  
Enter a number to be searched for:  
8  
8 found in position 3

#### Sample Dialogue 2

Enter a list of at most  
10 integers,  
and then press return:  
2 6 4 7 8 5  
Enter a number to be searched for:  
9  
9 is not on the list.

#### Sample Dialogue 3

Enter a list of at most  
10 integers,  
and then press return:  
  
Enter a number to be searched for:  
3  
3 is not on the list.

**Figure 9.10**  
(continued)



When this function call is executed in the first sample run of the program, the formal parameter *N* is set equal to the value of the actual parameter *Number*, and so it is set equal to 8. In exactly the same manner, the formal parameter *A* is set equal to the value of the actual parameters *List*, and so it is set equal to

10, 9, 8, 7, 6, 5, 4, 3, 2, 1

The entire array value is passed as a single unit.

Note that in the second and third dialogues, the array is allowed as a parameter even though it is not completely filled. However, the function should not attempt to access any of the undefined elements. In these cases, it does not, and so there are no problems.

---

## Pitfall

### Type Mismatches with Array Parameters

As we noted in the last chapter, two Pascal types are not the same unless they have the same name. This can cause problems if you use anonymous types when declaring arrays. Consider the following:

```
type FirstName = array[1..10] of integer;  
var A: array[1..10] of integer;
```

With these declarations, the array *A* cannot be used as an actual parameter of type *FirstName*. To avoid this problem, always declare a unique name for each array type, and always refer to it by that name.

---

## Efficiency of Variable Parameters

An array uses large amounts of storage compared to that used by a simple variable. The compiler typically allows one storage location for each indexed variable. For example, the array *List* in Figure 9.10 would normally consume ten times as much storage as a simple variable of type *integer*. There is no need to be obsessive about saving storage, but excessive use of storage can cause a program to run more slowly, or even to run out of storage and terminate abnormally. There are several things you can do to avoid excessive use of storage with arrays. The most obvious technique is to not declare arrays to be any larger than necessary. Another technique for saving storage has to do with procedure parameters.

Variable parameters typically consume less storage than do value parameters. The reason is that a value parameter is a local variable that is set to the value of the actual parameter. So if *A* is a formal value parameter of an array type and *List* is the corresponding actual parameter, then when the procedure is called, two arrays are in stor-

---

**Program**

```

program CountLetters(input, output);
{Counts the number of occurrences of each letter in a sentence.
Assumes the lowercase letters are contiguous.}
type Letter = 'a' .. 'z';
   LetterCounter = array[Letter] of integer;
var Count: LetterCounter;

procedure ReadSentence(var Count: LetterCounter);
{Sets Count['a'] equal to the number of 'a'-s in an input sentence;
Count['b'] equal to the number of 'b'-s, and so forth down to Count['z'].}
var Symbol: char;
begin{ReadSentence}
  for Symbol := 'a' to 'z' do
    Count[Symbol] := 0;

  while not eoln do
    begin{reading line}
      read(Symbol);
      if ('a' <= Symbol) and (Symbol <= 'z') then
        Count[Symbol] := Count[Symbol] + 1
      end; {reading line}
    readln
  end; {ReadSentence}

procedure DisplayCount(var Count: LetterCounter);
{Outputs the nonzero elements of the array Count.}
const Blank = ' ';
var Symbol: Letter;
begin{DisplayCount}
  for Symbol := 'a' to 'z' do
    if Count[Symbol] <> 0 then
      writeln(Count[Symbol], Blank, Symbol)
end; {DisplayCount}

begin{Program}
  writeln('Enter a sentence and press return. ');
  writeln('all lowercase letters, please. ');
  ReadSentence(Count);
  writeln('Your sentence contains: ');
  DisplayCount(Count)
end. {Program}

```

**Figure 9.11**  
**Program using**  
**'a' .. 'z' as**  
**an index type.**

### Sample Dialogue

```

Enter a sentence and press return.
all lowercase letters, please.
may the hair on your toes grow long and curly.
Your sentence contains:
  3 a
  1 c
  1 d
  2 e
  2 g
  2 h
  1 i
  2 l
  1 m
  3 n
  5 o
  4 r
  1 s
  2 t
  2 u
  1 w
  3 y

```

**Figure 9.11**  
(continued)

age: the global array `List` and the local array `A`. If, on the other hand, `A` is a variable parameter, then it is just a formal blank, which is filled in with `List`. In that case, there is only one array in storage. Hence, even if a procedure does not change an array, it makes sense to declare it as a variable parameter in order to save storage. For example, if the declaration heading for the function `Search` in Figure 9.10 is changed to the following, then the array parameter `A` will be a variable parameter, and so the program will use less storage.

```

function Search(N: integer; var A: IntegerArray;
               Last: ExtendedIndex): ExtendedIndex;

```

---

## Array Example with Noninteger Indexes

As we have already noted, the index type of an array need not be a subrange of the integers; it can be a subrange of any ordinal type. The program in Figure 9.11 illustrates the use of the subrange type `'a' .. 'z'`, both as an array index type and as the type of a `for` loop control variable. That program reads in a sentence and then uses an array indexed by `'a' .. 'z'` to count the number of occurrences of `'a'`, `'b'`, and so forth in the sentence. The program assumes that the lowercase letters are contiguous (there are no symbols between any two alphabetically consecutive lowercase letters). This assumption holds for most, though not all, systems.

---



---

## Random versus Sequential Array Access

The letter-counting program in Figure 9.11 illustrates the two principal methods for accessing array elements. In the procedure `ReadSentence`, the array elements of the array `Count` are initialized to zero by stepping through the array indexes in order. This method is called *sequential access*.

The other method of array access is used when the letters of the sentence are read in and counted. If the letter 'a' is read in, then the first array element is changed; if the letter 'z' is accessed, then the last array element is changed; if the letter 'd' is read, then the fourth element is changed. This is referred to as *random access*, since the order of the elements accessed cannot be determined beforehand.

---

## Array Indexes with Semantic Content

The indexes of an array allow us to access individual elements in an organized fashion. Often they serve no purpose other than that of an arbitrary numbering of the array elements, as in our first sample program with a list of five scores (Figure 9.4). Not infrequently, however, the array indexes can carry some meaning. For example, in our letter-counting program, the index 'a' is not arbitrary. It stands for the letter 'a', and the indexed variable `Count['a']` has a value equal to the number of times the letter 'a' has been seen. Choosing array indexes with semantic content can often simplify a program. In our letter-counting program, this allowed for random access to the array. The program did not have to search a list of letters to find the location for, say, 'm'; it went directly to `Count['m']` by using the index 'm'.

Integers are typically used when we need arbitrary indexes without any particular meaning. Often, they can also have meaning. In a list of students, they might serve as the students' numbers. In a list of checks, they may serve as the check numbers.

---

## Enumerated Types as Array Index Types (Optional)

Since they are ordinal types, enumerated types and subranges of enumerated types may be used as the index types of arrays. For example, the following declares `Sales` to be an array of integers indexed by an enumerated type for the days of the week:

```
type WeekDay = (Mon, Tue, Wed, Thur, Fri, Sat, Sun);  
    SalesList = array[WeekDay] of integer;  
var Sales: SalesList;
```

The array might be used to keep track of the number of automobiles sold by an auto dealer's sales force. Once the array has been filled, the weekly sales can be totaled as follows:

---

**Program**

```

program ShowSales(input, output);
{Displays a week's sales day by day and classifies
each day as above or below average.}
const Width = 6;
type WeekDay = (Mon, Tue, Wed, Thur, Fri, Sat, Sun);
   SalesList = array[WeekDay] of integer;
var Sales: SalesList;
    Day: WeekDay;
    Total: integer;
    Average: real;
begin{Program}
  writeln('Welcome to the sales meeting. ');
  writeln('Enter units sold for Monday through Sunday: ');
  for Day := Mon to Sun do
    read(Sales[Day]);

  Total := 0;
  for Day := Mon to Sun do
    Total := Total + Sales[Day];
  Average := Total/7;

  writeln('Daily sales compared to average: ');
  writeln('Mon':Width, 'Tue':Width, 'Wed':Width,
    'Thu':Width, 'Fri':Width, 'Sat':Width, 'Sun':Width);
  for Day := Mon to Sun do
    write(Sales[Day]: Width);
  writeln;
  for Day := Mon to Sun do
    if Sales[Day] > Average then
      write('+': Width)
    else if Sales[Day] < Average then
      write('-': Width)
    else {Sales[Day] = Average}
      write('0': Width);
  writeln;
  writeln('Go get 'em!')
end. {Program}

```

**Sample Dialogue**

```

Welcome to the sales meeting.
Enter units sold for Monday through Sunday:
6 7 10 9 8 5 11
Daily sales compared to average:
   Mon    Tue    Wed    Thu    Fri    Sat    Sun
     6      7     10      9      8      5     11
     -      -      +      +      0      -      +
Go get 'em!

```

**Figure 9.12**  
(Optional)

Enumerated type  
as an index type.

```

Total := 0;
for Day := Mon to Sun do
    Total := Total + Sales[Day]

```

The variable Day is of type WeekDay, and Total is of type integer.

Figure 9.12 shows these details embedded in a complete program that reads in daily sales and then echoes them back with each day marked as above average (+), below average (-), or average (0).

---

## Case Study

---

### Production Graph

#### Problem Definition

The Apex Plastic Spoon Manufacturing Company has commissioned you to write a program that will display a bar graph showing the productivity of each of their four manufacturing plants for any given week. Each plant keeps separate production figures for each department, such as the teaspoon department, soup spoon department, plain cocktail spoon department, colored cocktail spoon department, and so forth. Moreover, each plant has a different number of departments. For example, only one plant manufactures colored cocktail spoons. The input is entered plant by plant and consists of a list of numbers giving the production for each department in that plant. The output will give the total production for each plant in the form of a bar graph like the following:

```

Plant # 1 * * * * *
Plant # 2 * * * * *
Plant # 3 * * * * *
Plant # 4 * * * * *

```

Each asterisk represents 1000 units of output.

Notice that the output is in thousands of units, and hence the program must scale the output by dividing it by 1000. This presents a problem, since the computer must display some whole number of asterisks. It cannot display 1.6 asterisks for 1600 units. We will thus round to the nearest thousand, and so 1600 will be the same as 2000 and will produce two asterisks.

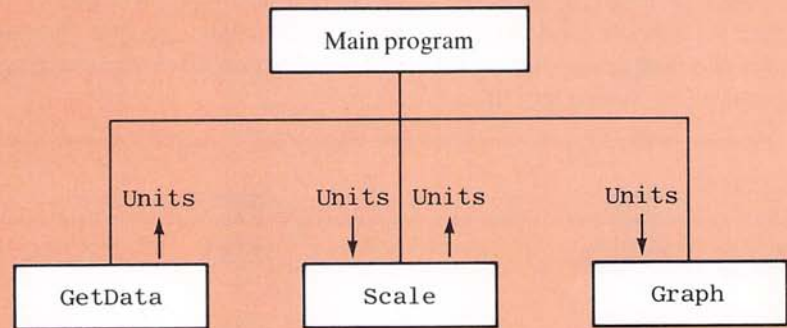
*refining the  
problem definition*

#### Discussion

We will use an array called Units that is indexed by plant numbers 1 . . . 4 and that will hold the total output of each plant. For example, Units[3] will be set equal to the total output of plant number three. Since the output is in thousands of units, the program will scale the values of the array elements. If the value of Units[3] is 2040, for example, it will be scaled to 2, and eventually two asterisks will be output. The task can be divided into the following subtasks:

1. GetData: For each plant I, read input and set Units[I] equal to the total production for plant number I.
-





Units: an array of type.

`array[1 . . <number of plants>]` of integer.

`Units[I]`: first set equal to the number of units produced by plant I and later set equal to the number of asterisks that will indicate this number of units on the bar graph.

**Figure 9.13**  
Data flow diagram  
for production  
graph.

2. Scale: For each I change the value of `Units[I]` to the correct number of asterisks for plant number I.
3. Graph: Output the bar graph.

The interaction of the subtasks is shown in the data flow diagram in Figure 9.13.

#### *GetData*

The program receives separate input figures for each department within a plant, but the output is in terms of the total output for a plant. Hence, the program must total the output of all departments in a plant in order to get the appropriate figure for that plant. These observations lead us to the following basic outline of the algorithm for `GetData`:

#### **ALGORITHM**

```

for I := 1 to <last plant number> do
  begin{for}
    read in all the data for plant number I,
    total the numbers, and
    set Units[I] equal to that total.
  end {for}

```

The *body* of the *for* loop sums a list of numbers and leaves the value of `Units[I]` equal to this sum. In other contexts we have designed code for just this task. If we wanted the sum stored in a variable called `Total` instead of being stored in `Units[I]`, then we would know how to proceed. To store a sum of input numbers in the variable `Total`, we would use the following:

```
Total := 0;
while not eoln do
  begin{while}
    read (Next);
    Total := Total + Next
  end {while}
```

*adapting  
another  
ALGORITHM*

If we instead want the total to be in `Units[I]`, all we need do is substitute `Units[I]` for `Total` and we will obtain code for the body of our *for* loop:

```
for I := 1 to <last plant number> do
  begin{for}
    Units[I] := 0;
    while not eoln do
      begin{while}
        read (Next);
        Units[I] := Units[I] + Next
      end; {while}
    readln
  end {for}
```

The final `readln` is needed in order to advance to the next input line so that the next iteration of the *for* loop starts at the beginning of a line.

In order to scale the numbers to an integral number of thousands, we want to divide by 1000 and then round to the nearest integer. Fortunately, both of these are built-in Pascal operations. To scale `Units[I]`, the following works:

*Scale*

```
Units[I] := round(Units[I]/1000)
```

To scale the entire array, we can use a *for* loop to do this to each element of the array. The final procedure declaration is given as part of Figure 9.14.

The algorithm for the procedure `Graph` is straightforward:

```
for I := 1 to <last plant number> do
  begin
    write('Plant #', I);
    repeat the following Units[I] number of times
      write('*');
    writeln
  end
```

*Graph  
ALGORITHM*

The “repeat *n* number of times” loop can be implemented in the standard fashion using a *for* loop. The complete procedure, included in the complete final program, is given in Figure 9.14.

**Program**

```

program ShowProduction(input, output);
{Reads data for each plant in the company and displays a
bar graph showing productivity of each plant.}
const NumPlants = 4;
type Index = 1 .. NumPlants;
    List = array[Index] of integer;
var Units: List;

procedure GetData(var Units: List);
{Postcondition: Units[I] contains the number of units produced by plant I.}
var I: Index;
    Next: integer;
begin{GetData}
    for I := 1 to NumPlants do
        begin{for}
            writeln;
            writeln('Enter number of units produced for');
            writeln('each department in plant number', I);
            writeln('End by pressing return. ');
            Units[I] := 0;
            while not eoln do
                begin{while}
                    read(Next);
                    Units[I] := Units[I] + Next
                end; {while}
            readln
        end {for}
    end; {GetData}

procedure Scale(var Units: List);
{Changes the values of Units[I] so that
it records 1000s of units. Rounds to the nearest
1000, e.g., 5020 is changed to 5, but 5900 is changed to 6.}
var I: Index;
begin{Scale}
    for I := 1 to NumPlants do
        Units[I] := round(Units[I]/1000)
    end; {Scale}

procedure Graph(Units: List);
{Displays a bar graph showing production of each plant.
Precondition: Units[I]*1000 is productivity of plant I to the nearest 1000.}
var I: Index;
    J: integer;

```

**Figure 9.14**  
**Production graph**  
**program.**



```

begin{Graph}
  writeln;
  writeln('Units produced in thousands of units:');
  for I := 1 to NumPlants do
    begin{Outer loop}
      write('Plant #', I:2);
      for J := 1 to Units[I] do
        write('*');
      writeln
    end {Outer loop}
  end; {Graph}

begin{Program}
  writeln('This program displays a graph showing');
  writeln('production for each plant in the company. ');
  GetData(Units);
  Scale(Units);
  Graph(Units)
end. {Program}

```

### Sample Dialogue

This program displays a graph showing  
production for each plant in the company.

Enter number of units produced for  
each department in plant number 1  
End by pressing return.

2000    3000    1000

Enter number of units produced for  
each department in plant number 2  
End by pressing return.

2050    3002    1300

Enter number of units produced for  
each department in plant number 3  
End by pressing return.

5000    4020    500    4348

Enter number of units produced for  
each department in plant number 4  
End by pressing return.

2507    6050    1809

Units produced in thousands of units:

Plant# 1\*\*\*\*\*

Plant# 2\*\*\*\*\*

Plant# 3\*\*\*\*\*

Plant# 4\*\*\*\*\*

**Figure 9.14**  
(continued)

## Off-Line Data (Optional)

Arrays are often used to process large amounts of data. If the data set is very large, then it is impractical to enter the data interactively from the keyboard. For example, suppose that in the previous case study each plant had hundreds of departments instead of just a few. In that case, it would make sense to read the data off-line in the manner discussed in the optional section of Chapter 7 entitled “Off-Line Data and a Preview of EOF.” The general method of reading in the data would be the same, except that there is no point in outputting instructions to the user. The data set is prepared before the program is run and must be in a format that matches what the program expects. If it does not match, then either the data must be reformatted or the program must be changed to accommodate the data.

---

## Case Study

---

### Sorting

#### Problem Definition

One of the most commonly encountered programming tasks, and certainly the one most thoroughly studied, is that of sorting a list of values. For example, the list might be a list of exam scores, and we may want to see them sorted from lowest to highest or from highest to lowest; the list might be a list of words that we have misspelled, and we may want to see them in alphabetical order. In this section we will consider lists of integers and design a procedure that sorts a list into the order smallest to largest. The list will be stored in an array of the following type, where `Low` and `High` are defined constants.

```
type Index = Low .. High;  
List = array[Index] of integer;  
var A: List;
```

Since we want to accommodate partially filled arrays, we will use a variable called `Last` to record the last array index used, and we will write our procedure to sort the elements `A[Low]` through `A[Last]` and ignore array locations with indexes greater than `Last`.

#### Discussion

*first  
design idea*

One way to design an algorithm is to rely on the definition of the problem. In this case, the problem is to sort an array, such as `A`, from smallest to largest. This means re-

---

arranging the values so that  $A[1]$  is the smallest,  $A[2]$  is the next smallest, and so forth. The definition yields an outline for a straightforward algorithm:

```
for I := Low to Last do
  put the Ith smallest element into A[I]
```

There are many ways to realize this general approach. The details can be developed using two arrays and copying the elements from one array to the other in sorted order. However, one array should be both adequate and economical, and so we decide to develop the algorithm using only one array. To help in exploring the problem, we will use the example given in Figure 9.15 as the original array value, and we will try sorting that array using pencil and eraser. (In Figure 9.15, the array is full, and so  $\text{Last} = \text{High} = 10$ .) When we search the array to find the smallest element, we discover that it is the value of  $A[4]$ , which is 2. We next want to set  $A[1]$  equal to this value of 2. However, in doing so we must be careful to not lose the original value of  $A[1]$ , which is 8. A simple assignment statement like the following would destroy the 8.

```
A[1] := A[4] {No good; destroys the original value of A[1].}
```

Now that we understand some of the problems involved in this example, we can begin to formulate a strategy for handling them.

We wish to set the value of  $A[1]$  equal to the value of the index variable with the smallest value; in the example, that is the value of  $A[4]$ . When we do this we must preserve the old value of  $A[1]$  so that it can be inserted into the array at some later time. This is illustrated in the second “snapshot” of the array shown in Figure 9.15. The algorithm must do something with the displaced value 8 that was the original value of  $A[1]$ . Fortunately we have a vacant array location in which to store it. Since the value 2 has “left”  $A[4]$ , the algorithm can place the 8 there. In other words, the values of  $A[1]$  and  $A[4]$  are simply interchanged. A similar thing is done with  $A[2]$ , as shown in the figure. The entire array can be sorted by a series of interchanges such as these two. Any sorting algorithm that is based on interchanging elements is referred to as an *interchange sort*.

The simplest interchange sorting algorithm is called *selection sort* and is the one we obtain by proceeding as we did with our sample array values. In outline form it is as follows:

```
for I := Low to Last - 1 do
  Exchange (A[I], A[<suitable index>])
```

( $A[\text{Last}]$  will automatically be in place once the previous elements are sorted.)

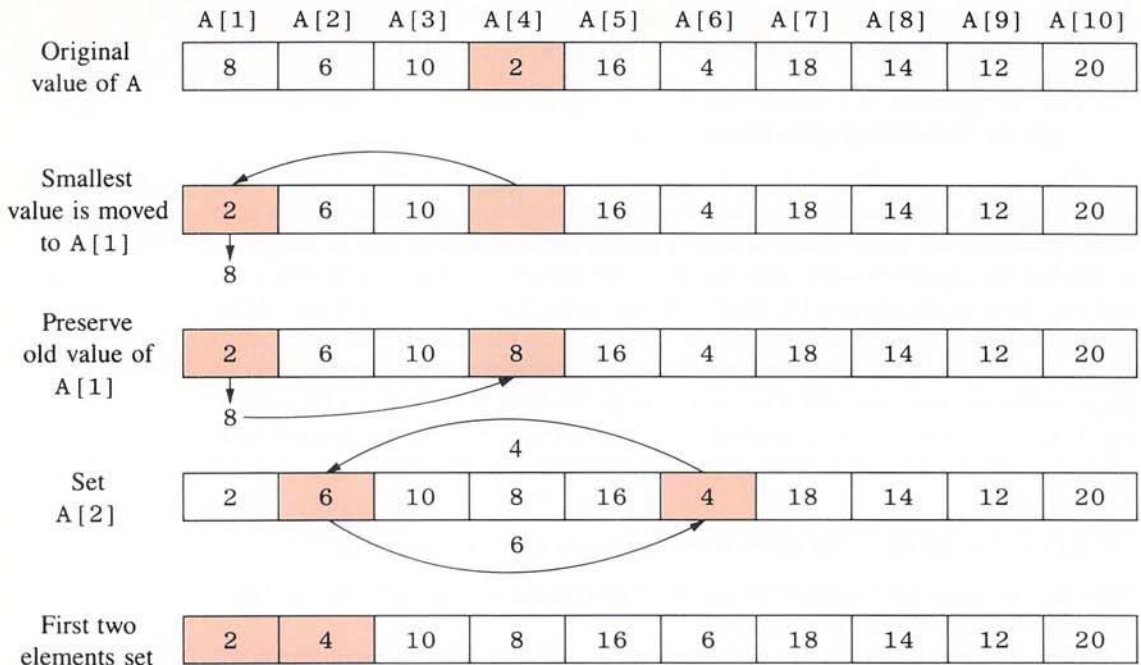
All that remains is to calculate the expression  $\text{<suitable index>}$ . When the loop considers  $A[I]$  and looks for a suitable interchange, the indexed variables with indexes smaller than  $I$  already contain the correct values for a sorted array. So the sought-after index is the index of the smallest of the remaining elements  $A[I]$ ,  $A[I + 1]$ , ...,  $A[\text{Last}]$ . Since this index is a single value, we can define a function to return the index. The value of  $\text{<suitable index>}$  will be  $\text{Imin}(A, I, \text{Last})$ , where  $\text{Imin}$  is as declared in Figure 9.16 on page 366. The complete sorting algorithm is implemented by the procedure `Sort`, which is also declared in Figure 9.16.

*exploring  
the problem*

*interchange  
sorting*

**ALGORITHM**  
*selection  
sort*





**Figure 9.15**  
Interchange  
sorting.

---

Never trust to general impressions, my boy,  
but concentrate yourself upon details.  
*Sir Arthur Conan Doyle (Sherlock Holmes),  
A Case of Identity*

---

## Summary of Problem Solving and Programming Techniques

- An array type can be used to produce a unified naming scheme for a collection of related values.
- A big advantage of array types is that by manipulating the array index a program can actually compute a name for an (indexed) variable.

- An array indexed variable can be used anywhere that a simple variable of the array's component type can be used.
- In Pascal the size of an array must be declared at the time the program is written. Hence, if the desired size will vary from one run of the program to another, the array must be declared to be of the largest size possible. In such cases, the program must keep track of which indexed variables are actually used.
- It is a good idea to use defined constants for one or both bounds of an array index type. That way the array size can be changed simply by changing the constant declarations.
- A common bug in programs that use arrays is attempting to use an index value outside the defined range for an array's indexes.
- One often needs a variable that takes on values one step beyond the index range of an array. In such cases, do not declare the variable to be of the index type. Instead declare it to be of a larger subrange type or of the host type (such as `integer`.)
- A `for` loop is a natural way of proceeding sequentially through an array.
- The "brute force" method of looking at every element of an array can be used to search an array for almost any property. It can be inefficient, but it is simple and effective.
- One way to design an algorithm is to think about how you would solve the problem using pencil and paper, and then design the algorithm to do that or some variation of it.
- Array indexes can carry information that serves to identify the corresponding elements of the array. For example, the indexes might be student numbers or check numbers or the letters of the alphabet.
- Array elements may be accessed either sequentially, as with a `for` loop, or in a "random" order by computing the desired index.
- There are two different ways of viewing an array: as a collection of (indexed) variables of the component type, and as a single variable with a single (compound) value consisting of a list of values of the component type. Sometimes it is more productive to take one view; other times the alternative view is more productive.
- An array variable without subscripts may be used in an assignment statement, like so: `A := B`.
- Either a single array element or an entire array can be passed as a parameter to a procedure. In the first case, the formal parameter type must be the component type of the array; in the second case, it must be the array type.
- If an indexed variable is used as an actual parameter to a procedure, the index expression is evaluated before the actual parameter is substituted for the formal parameter.
- It is good practice to declare a name for each array type definition (or any other type definition) and to use the type name, rather than the type definition, when declaring array variables. Both formal and actual parameters of an array type must always be specified by a type name.
- Arrays consume large quantities of storage, and some care should therefore be taken not to use excessive storage when dealing with arrays. One storage-conserving technique is to use variable rather than value parameters for array type parameters.

**Program**

```

program SortTest(input, output);
{Tests the procedure Sort.}
const Low = 1;
      High = 10;
      LowMinus1 = 0; {Low - 1}
type Index = Low .. High;
      List = array[Index] of integer;
      ExtendedIndex = LowMinus1 .. High;
var A: List;
    I: Index;
    Last: ExtendedIndex;

procedure Exchange(var X, Y: integer);
{Interchanges the values of X and Y.}
var Temp: integer;
begin {Exchange}
    Temp := X;
    X := Y;
    Y := Temp
end; {Exchange}

function Imin(var A: List; Start, Last: Index): Index;
{Returns the index I such that A[I] is the smallest of the values: A[Start], A[Start + 1], ..., A[Last].}
var Min, I: integer;
begin {Imin}
    Imin := Start; {tentatively}
    Min := A[Start]; {Minimum so far.}
    for I := Start + 1 to Last do
        if A[I] < Min then
            begin {then}
                Min := A[I];
                Imin := I
                {Min is the smallest of the values A[Start], ..., A[I];
                 the tentative value of Imin is x such that A[x] = Min.}
            end {then}
    end; {Imin}

procedure Sort(var A: List; Last: ExtendedIndex);
{Sorts the partially filled array A into increasing order using the
selection sort algorithm. Postcondition: The array elements have been
rearranged so that A[Low] <= A[Low + 1] <= ... <= A[Last]}
var I: Index;

```

**Figure 9.16**  
Selection sort.



```

begin{Sort}
  for I := Low to Last - 1 do
    begin{Place correct value in A[I]}
      Exchange (A[I], A[Imin (A, I, Last) ])
      {A[Low] ≤ A[Low + 1] ≤ . . . ≤ A[I] are the smallest of the original array
       elements; the remaining elements are in the remaining positions.}
    end {Place correct value in A[I]}
end; {Sort}

begin{Program}
  writeln('Enter a list of numbers. ');
  writeln('I will take up to ', High - Low + 1, ' numbers. ');
  Last := LowMinus1;
  while not eoln do
    begin{while}
      Last := Last + 1;
      read(A[Last])
    end; {while}

  Sort(A, Last);

  writeln('In sorted order the numbers are: ');
  for I := Low to Last do
    write(A[I]);
  writeln
end. {Program}

```

#### Sample Dialogue

```

Enter a list of numbers.
I will take up to 10 numbers.
80 10 50 70 60 90 20 30 40
In sorted order the numbers are:
10 20 30 40 50 60 70 80 90

```

**Figure 9.16**  
(continued)

---

## Summary of Pascal Constructs

### type declaration

Syntax:

```

type <type name 1> = <type definition 1>;
    <type name 2> = <type definition 2>;
    .
    .
    .
    <type name n> = <type definition n>;

```

---

Example:

```
type Index = 0 .. 100;  
List = array[Index] of char;
```

The type names are identifiers chosen by the programmer. The type definitions can be any of the type definitions described in this book. The type declaration section of a block comes after the constant declarations and before the variable declarations. The identifier *type* is used only once, even if there is more than one type definition.

### array types

Syntax:

```
array [<index type>] of <component type>
```

Examples:

```
array[0 .. 100] of integer  
array[Index] of char
```

Form of an array type definition. <index type> must be either the definition of a subrange type or the name of an ordinal type, such as the name of a defined subrange type or a predefined ordinal type like *char*. The type *integer* cannot be used as an <index type>. The <component type> can be any Pascal type.

### array declaration

Syntax:

```
var <array variable name> : <array type name>;
```

Example:

```
var A: List;
```

The way to declare an array variable. The declaration for an array variable is just like the declaration for any other type of variable. Note that the array variable is declared without any square brackets or indexes. An array type definition may be used in place of an array type name, but it is usually preferable to use a defined type name. (The sample array type *List* is defined in the first entry of this summary.)

### indexed variable

Syntax:

```
<array name> [<index expression>]
```

Example:

```
A[I + 1]
```

An indexed variable of the array <array name>. The <index expression> can be any expression that evaluates to a value whose type is the index type of the array. An indexed variable can be used anyplace that a variable of the component type of the array can be used.

---

## Exercises

### Self-Test Exercises

10. Write suitable array type declarations for each of the following:
  - a. An array to record the number of students who received a grade of 1 on a quiz, the number who received a grade of 2, and so forth, up to the maximum grade of 10.
  - b. An array to record the number of children in a school for each age from 5 to 13.
  - c. An array to hold the amount of each check you write this month, recorded so that you can deduce the amount given the number of the check. Check numbers range from 661 to 753.
11. (This exercise uses the optional section “Enumerated Types as Array Indexes.”) Write two suitable enumerated type definitions, one for all the days of the week and one for the days Monday through Friday. Also write two array type declarations, one for an array to record the hours worked on each day from Monday through Friday and one for the number of hours reserved for play on each day of the week, including Saturday and Sunday.
12. Give a simpler way of accomplishing the following:

```
for I := 0 to 10 do
  A[I] := B[I]
```

A, B are of type *array*[0 . . 10] of integer.

13. The following code is supposed to set *Found* equal to *true* if *N* is in the array *A* and to *false* otherwise. It does not work correctly. What is wrong?

```
Found := false; {tentatively}
for I := Low to Last do
  if N = A[I] then
    Found := true
  else
    Found := false
```

14. Write a procedure that reverses the order of the elements in an array of the following type:

```
type Word = array[0 . . 100] of char;
```

15. What changes do you need to make to the procedure *Sort* in Figure 9.16 so that it sorts numbers in the order largest to smallest rather than smallest to largest?
16. What changes do you need to make to the procedure *Sort* in Figure 9.16 so that it sorts a list of numbers of type *real* rather than a list of numbers of type *integer*?
17. What changes do you need to make to the procedure *Sort* in Figure 9.16 so that it sorts a list of lowercase letters into alphabetical order instead of sorting a list of numbers into numeric order?



### Interactive Exercises

18. Run the following two programs to see what error messages your system gives. TURBO Pascal users try them both with and without the R compiler directive set.

```
program IndexProblem(input, output);
var A: array[1 . . 10] of integer;
begin
  A[0] := 1 {Array index is too small.}
end.

program VariableProblem(input, output);
var X: 1 . . 10;
begin
  X := 0 {Subrange variable assigned too small a value.}
end.
```

19. Write a program that allows the user to type in up to 10 positive numbers and then echoes back the numbers typed in but in reverse order. Have the user terminate the list with a negative number. The answers to Exercises 9 and 14 might be of some help.

### Programming Exercises

20. Write a program that reads 10 integers into an array, computes the average, largest, and smallest numbers in the array, and finally outputs the numbers plus the amount by which each one differs from the smallest, the largest, and the average.

21. Write a program to keep a budget. There are five numbered budget categories: 1 for food, 2 for housing, 3 for clothing, 4 for utilities, and 5 for entertainment. The program will display the categories, showing the number of each category, and then the user and the program will refer to categories by number rather than by name. The user may name any category by number and then enter an amount. The program records the amount spent and keeps track of the total amount spent in each category. When the user indicates that he or she wants to see the totals, the program prints out the amount spent in each category and the total amount spent in all categories combined. The user may enter more amounts after seeing the total and may ask for a new total later on.

22. Write a program that computes grades for a class of up to 50 students. The program reads in a score in the range 0 . . 100 for each student and then outputs the grades, identifying each student by number. The first student read in is student number 1, the next is student number 2, and so forth. Grades are to be determined as follows. Any student who receives 10 points below the average receives an F. Any student who receives a score above that and at most 10 points above the average receives a C. Any student who receives a score above that receives an A. There are no D's or B's. To use your program for larger classes, you should not need to change anything except the constant declaration section.

23. Write a program to read in four letters and then output all 24 permutations of these letters. Use an array to hold the four letters. Do not cut corners in designing this one; it can be confusing.

---

- 
24. Write a program to play Nim using five piles of sticks. Use an array to store the sizes of the piles. The game is explained in Exercise 27 of Chapter 8.
25. (This exercise uses the optional sections of Chapter 8 on pseudorandom number generators.) Write a procedure declaration for a procedure called `Deal` that sets the value of a variable parameter to a value that represents a card chosen at random from a standard 52-card deck. The procedure should also keep track of the cards already dealt out, so that it does not deal a card twice. For example, after dealing out four aces, it should not deal out a fifth ace at any later time. Use an array parameter to keep track of the cards already dealt out. There is no need to keep track of suits (clubs, hearts, etc.).
26. Write a program that reads in two lists of 10 or fewer numbers, each into one of two arrays. The input is assumed to be in numeric order. The program will then output a list of all the (up to twenty) numbers in numeric order. This is called *merging* the lists.
27. Make the unrealistic assumption that your computer has `maxint` equal to 18 and write procedures to read in two numbers of 10 or fewer digits and output their sum. The numbers are to be read as character values one digit at a time, using a variable of type `char`, and then converted into numbers and stored in two arrays of 10 integer values each. If the number is less than 10 digits long, leading zeros should be added to the array. An addition procedure is then called to calculate the sum and store it in a third array. If the sum would be more than 10 digits long, the program issues an “integer overflow” message. If the sum is less than 10 digits long, the two numbers read in and their sum are output.
- 

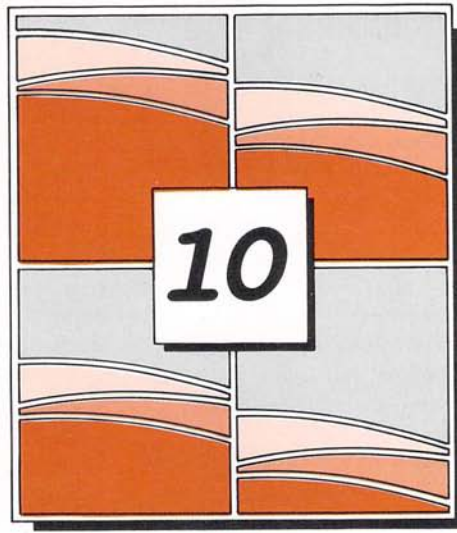
## References for Further Reading

The following books contain more information on sorting:

- P. Helman and R. Veroff, *Intermediate Problem Solving and Data Structures—Walls and Mirrors*, 1986, Benjamin/Cummings, Menlo Park, Ca.
- B.W. Kernighan and P.J. Plauger, *Software Tools in Pascal*, 1981, Addison-Wesley, Reading, Mass.
-







## ***Complex Array Structures***

Memory is necessary  
for all the operations of reason.  
*Blaise Pascal*

## Chapter Contents

Strings of Characters	Standard Pascal—Packed Arrays (Optional)
Arrays of Arrays	Standard Pascal—Packed Arrays of Characters (Optional)
TURBO Pascal—The Types Array of Char and String	Self-Test Exercises
Parallel Arrays	Multidimensional Arrays
TURBO Pascal Case Study—Making a History Table	Case Study—Grading Program
Self-Test Exercises	Pitfall—Exceeding Storage Capacity
The Notion of a Data Structure	TURBO Pascal—Typed Constants (Optional)
Data Abstraction	TURBO Pascal—Array Constants (Optional)
TURBO Pascal—Clearing the Screen	Summary of Problem Solving and Programming Techniques
TURBO Pascal Case Study—Automated Drill	Summary of Pascal Constructs
Adapting a Known Algorithm	Exercises
Standard Pascal Case Study—Pattern Matching	References for Further Reading
Design by Concrete Example	

Like other structured types, which we will introduce in succeeding chapters, arrays may be declared so that they form a hierarchy. In this case, the hierarchy consists of an array of arrays. Other ways of structuring data involve the way we use arrays rather than the way in which the array type is defined. Two or more arrays, or an array and one or more variables, may form a conceptual unit for handling a particular kind of data. These data types and combinations of data types are called *data structures* and are the topic of this chapter. Most of these topics require no new Pascal constructs, but they do represent significantly new ways of reasoning about the tools we already have available. We do introduce one new Pascal concept in this chapter, namely the multidimensional array, which is an array whose indexed variables have two or more indexes instead of just a single index. We open with an important application using one of the simple array types introduced in the last chapter.

## Strings of Characters

One way to represent a string of characters, such as a word or a name or a line of text, is as an array of characters, such as the array variable `Line` declared as follows:

```
const MaxLength = 20;
type CharString = array[1..MaxLength] of char;
var Line: CharString;
    I: 1..MaxLength;
```

To read in a name typed at the key board, you can use:

```
for I := 1 to MaxLength do
  read(Line[I])
```

The name can be written out with the following similar statement

```
for I := 1 to MaxLength do
  write(Line[I])
```

There is one annoying feature of processing strings in this way: The string input must be exactly as long as the array. There must be one character for each indexed variable. For our sample declarations, the string of characters must always be exactly 20 characters long. If it is not, the user must type in extra blanks to fill the extra indexed variables. Filling the extra array positions with blanks is called *padding with blanks*. Typing in the extra blanks is burdensome to the user. A better alternative is to have the program fill the extra position with blanks. In Figure 10.1 we present a procedure to do just that, and we also present a procedure to write out an array of characters.

*padding  
with blanks*

---

## Arrays of Arrays

The component type of an array may be any Pascal type whatsoever. In particular, it can be an array type. Thus, we can have an array of arrays. A common situation that requires an array of arrays is an array of strings. The array variable `Name` when declared as follows, can be used to keep a list of names:

```
type CharString = array[1..MaxLength] of char;
    Index = 1..MaxI;
    StringList = array[Index] of CharString;
var Name: StringList;
```

`MaxLength` and `MaxI` are defined constants. Figure 10.2 illustrates a list of names stored in the array `Name`. In that figure `MaxLength` is set equal to 20.

With arrays of arrays, such as these arrays of strings, the notation can get a bit complicated, but there are no new rules involved. Just read or write the expressions

*nested  
indexes*



**Program**

```

program TestProcedures(input, output);
{Tests the procedures StringReadln and StringWriteln.}
const MaxLength = 20;
type CharString = array[1..MaxLength] of char;
var Line: CharString;
    I: 1..MaxLength;
    Ans: char;

procedure StringReadln (var A: CharString);
{Fills the array A with one line of input characters and fills any extra positions with blanks.
If there are too many characters it discards all characters after the first MaxLength characters.}
const Blank = ' ';
var I, J: integer;
begin{StringReadln}
    I := 0;
    while (not eoln) and (I < MaxLength) do
        begin{while}
            I := I + 1;
            read(A[I])
        end; {while}
    if not eoln {and hence I >= MaxLength} then
        writeln('Only ', MaxLength, ' characters read in. ');
    readln;

    {The entire string has been read in (discarding any characters beyond the
    first MaxLength characters.) The value of I is the last index used.
    If I = MaxLength then, the array A is full.}

    for J := I + 1 to MaxLength do
        A[J] := Blank
    end; {StringReadln}

procedure StringWriteln (var A: CharString);
{Writes the string A to the screen and advances to next line.}
var I: 1..MaxLength;
begin{StringWriteln}
    for I := 1 to MaxLength do
        write(A[I]);
    writeln
end; {StringWriteln}

```

**Figure 10.1**  
**Reading and**  
**writing strings of**  
**characters.**

```

begin{Program}
  repeat
    writeln('Enter a string and press return. ');
    for I := 1 to MaxLength do
      write(I mod 10:1); {To make it easy to count.}
    writeln;
    StringReadln(Line);
    writeln('The string array contains: ');
    StringWriteln(Line);
    writeln('More? (y/n) ');
    readln(Ans)
  until (Ans = 'n') or (Ans = 'N');
  writeln('End of test. ')
end. {Program}

```

### Sample Dialogue

```

Enter a string and press return.
12345678901234567890
Do Be Do
The string array contains:
Do Be Do
More? (y/n)
y
Enter a string and press return.
12345678901234567890
Life is a jelly doughnut. Isn't it?
Only      20 characters read in.
The string array contains:
Life is a jelly doug
More? (y/n)
n
End of test.

```

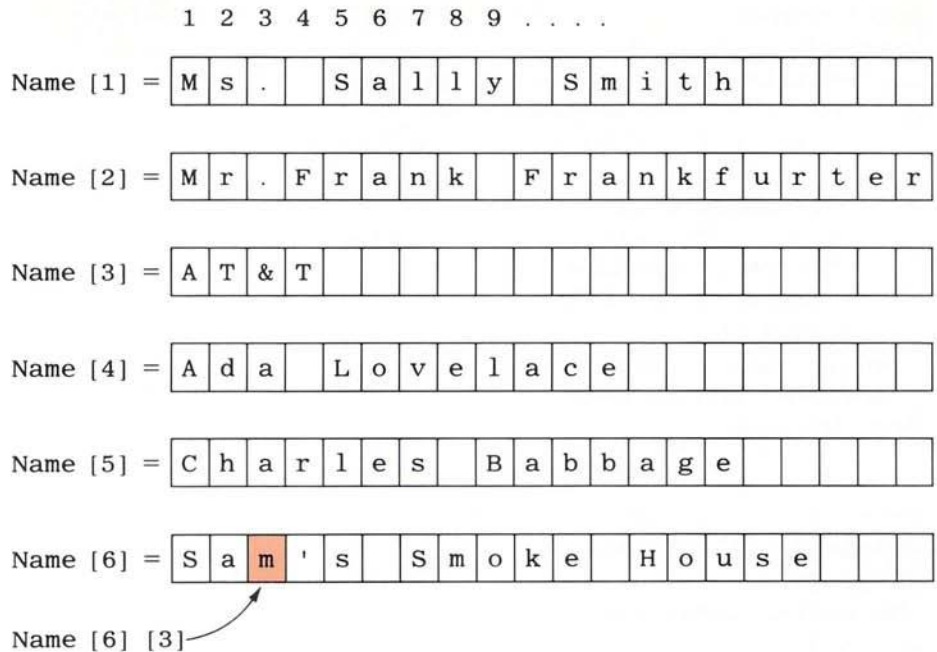
**Figure 10.1**  
(continued)

from left to right carefully and with courage. For example, consider the following expression:

Name[6][3]

Name[6] is an indexed variable of type CharString and in our example would contain the sixth name on our list of names. The type CharString is an array of characters, and so it makes sense to add an index to it. By adding [3] we obtain the third indexed variable of this array of characters. If we put this all together, we see that the above expression designates the location of the third letter in the sixth name on our list.

Expressions such as the one we just discussed usually need not be used at all. Instead it is preferable to think of Name[6] as a unit and to manipulate it as a unit using



**Figure 10.2**  
An array of  
strings.

procedures. For example, the following procedure call reads a string from the keyboard and stores it in the array Name [6]. (The procedure declaration is in Figure 10.1.)

```
StringReadln (Name [6])
```

It is true that when Name [6] is substituted for the formal parameter A in the procedure declaration it will produce expressions such as

```
Name [6] [I]
```

However, you need never think about this expression. The procedure StringReadln reads a string into an array of type CharString, and Name [6] is an array of that type. The procedure takes Name [6] as a single unit, gives it a value consisting of a string of characters, and then returns it to the calling program or procedure as a single unit. Once the procedure is written, you need not be aware of the details of how it accomplishes its task. You need not even be aware of the existence of that index I. That is the beauty of procedural abstraction.



## TURBO Pascal

### The Types Array of Char and String

Since TURBO Pascal has *string* types, it may seem pointless to use arrays of characters to do string processing in TURBO Pascal. This is usually correct, and, in the next two case histories, we will use the TURBO Pascal string-handling facilities, rather than arrays of characters. However, you should not completely ignore arrays of characters. In standard Pascal the special TURBO string-handling facilities are not available. If you move to a standard Pascal system, you will need to handle strings as arrays of some sort.

The relationship between *string* types and arrays of characters is very intimate. String variables in TURBO Pascal are implemented as something like arrays of characters and, as we saw in Chapters 2 and 8, the individual characters in a string variable can be accessed as if the string variable were an array of characters. If *Word* is a variable of type *string*, then *Word* [*I*] is an indexed variable whose value is the *I*th character in the value of *Word*. Hence, most of the techniques you learn for arrays of characters can also be used for *string* variables.

### Parallel Arrays

Oftentimes one wants the equivalent of two different component values for each indexed variable of an array. There is more than one way to accomplish this in Pascal. We will discuss other ways to do this later, but for now we will describe one standard and easy way to do it: Simply use two arrays with the same index type.

For example, if we need to record both a numeric score and a letter grade for each student in a class, then we can use the following declarations:

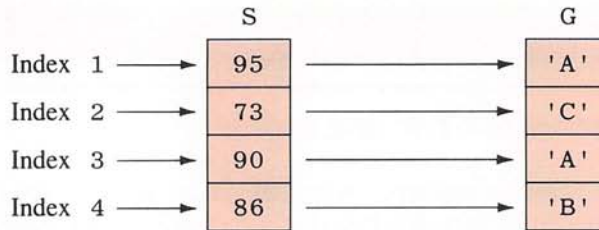
```
type ScoreList = array[1..Max] of integer;
     GradeList = array[1..Max] of char;
var S: ScoreList;
    G: GradeList;
```

The indexes serve as the link between the two types of grades. *S* [4] is the numeric score for student number four and *G* [4] is the letter grade for student number four. This is depicted in Figure 10.3. If *Last* is the number of the last student, then the following will output the list of students along with their scores and grades:

```
for I := 1 to Last do
  writeln('Student #', I, ' Score = ', S[I], ' Grade = ', G[I])
```

The two techniques of using parallel arrays and arrays of arrays can be combined. For example, if we are writing a program to act as a checkbook record keeper, we might use one array to keep track of the amount of each check and another array to

**Figure 10.3**  
Parallel arrays.



keep track of whom each check was made out to. If we let the array indexes be the check numbers, then we can use the following declarations:

```
type Index = 1 .. MaxI;
   AmountList = array[Index] of real;
   CharString = array[1 .. MaxLength] of char;
   StringList = array[Index] of CharString;
var Amount: AmountList;
    Name: StringList;
```

Here again, the indexes serve as the link between names and amounts. Name [7] and Amount [7] are joined by the index 7; they tell us to whom check number seven was made out and for what amount.

---

## TURBO Pascal Case Study

---

### Making a History Table

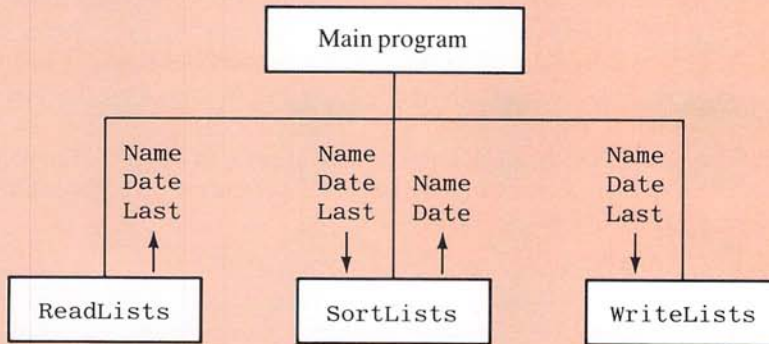
#### Problem Definition

Your music history text continually names composers and dates but has no table of composers and dates so that you can get a feel for the historical progression in one quick glance. We will design a program to solve this problem. Our program will accept input consisting of composers' names, followed by their years of birth. The program will output the same list sorted by birth date from the earliest to the most recent composer.

#### Discussion

*data  
representation*

The first step is to decide on a data representation. Now that we have array structures such as arrays of arrays and parallel arrays to choose from, this is no longer a straightforward task of naming a few simple variables. In this case the data will be a list in which each entry consists of a composer's name represented as a value of a *string* type and the composer's year of birth represented as an integer. It sounds like an array would be suitable, except that the data is of two different types: a string and an integer. In such cases, one natural solution is to use parallel arrays. That solution leads us to the following choice of data types:



### Data Summary

Name: an array of strings representing names.

Date: an array of integers representing years. Composer

Name [I] was born in the year Date [I].

Last: last index used in the arrays. (It will be the same for both arrays.)

**Figure 10.4**  
Data flow diagram  
for history table  
algorithm.

```

type NameString = string[MaxLength];
  Index = 1 .. MaxI;
  NameList = array[Index] of NameString;
  DateList = array[Index] of integer;
var Name: NameList;
    Date: DateList;
  
```

For each index  $I$ , Name [I] will contain the name of a composer, and Date [I] will contain his or her year of birth. We do not know the exact size of the list, so we will use a variable Last to record the last index used.

As shown in the data flow diagram in Figure 10.4, the task breaks down in a typical way into the three subtasks: input the data, sort the arrays, and give the output. The input can be handled in a straightforward manner using a loop that reads each composer's name and birth date. The output can be handled by another straightforward loop. The task of sorting the arrays is more complicated, and we now turn our attention to it.

It is easy to find natural-sounding subtasks for a problem, but these natural-sounding decompositions can sometimes be fruitless. Our sorting task has two arrays to sort. One natural-sounding breakdown into subtasks is the following:

*false  
subtasks*

1. Sort the array of dates.
2. Sort the array of names.



A serious problem arises with this approach. The dates can be sorted with little trouble, but we have no way to sort the names unless we sort them along with the dates. We want the names sorted by date of birth, but the dates are in the other array. These two subtasks are not separable. We must sort the two arrays together, sorting by date and moving the names to follow the dates. We must abandon the idea of solving two separate sorting subtasks, and instead we must treat the sorting of the two arrays as a single task.

In Chapter 9 we designed an algorithm to sort an array of integers. The procedure is in Figure 9.16. We will adapt that procedure to our present task. The procedure heading for our present sorting problem is

```

procedure SortLists (var Name: NameList;
                     var Date: DateList; Last: Index);
{Sorts the array Date into increasing order and moves the names to follow the dates.
Postcondition: The array elements in positions 1 . . Last of Date have been
rearranged so that Date[1] <= Date[2] <= . . . <= Date[Last];
if Date[I] was moved to Date[J], then Name[I] was moved to Name[J].}

```

A procedure similar to Sort in Figure 9.16 can be used to sort the array Date. We will need to adapt the algorithm used by that procedure so that it sorts both the array Date and the array Name. The algorithm applied to a single array such as Date is the following:

```

for I := 1 to Last - 1 do
    Exchange (Date[I], Date[<suitable index>])

```

For  $I = 1$  the <suitable index> is the index of the smallest date, for  $I = 2$  it is the index of the next smallest date, and so forth. The function Imin, which is also given in Figure 9.16, can be used to compute the <suitable index>, just as it was used in that program. (In this version of Imin, the array type is named DateList rather than List as it was in Figure 9.16.)

To adapt this sorting algorithm to the two parallel arrays in our problem, we need only move the names to follow the dates:

```

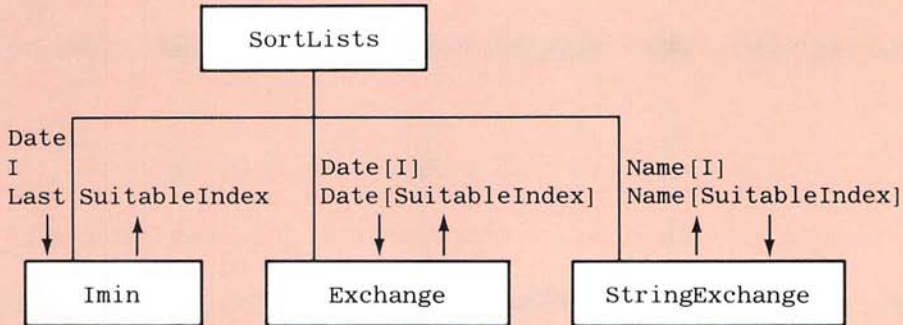
for I := 1 to Last - 1 do
    begin{Set Ith entries}
        Exchange (Date[I], Date[<suitable index>]);
        interchange Name[I] and Name[<suitable index>]
    end {Set Ith entries}

```

The procedure Exchange is well known to us by now. We need another procedure to accomplish a similar task for Name[I] and Name[<suitable index>]. Let us call this new procedure StringExchange. By way of summary, Figure 10.5 contains the data flow diagram for the procedure SortLists.

The procedure StringExchange must interchange the values of the two string variables Name[I] and Name[<suitable index>]. This task is essentially the same as interchanging the two integer dates. Hence, the procedure StringExchange can be identical to the procedure Exchange except that the type of the variables is NameString rather than integer. The detailed code for the procedure StringExchange is given in Figure 10.6 along with the rest of the program. Using the pro-

**ALGORITHM**  
*adapting to  
parallel arrays*



### Data Summary

Name: an array of strings representing names.

Date: an array of integers representing years. Composer Name [ I ] was born in the year Date [ I ] .

Last: last index used in the arrays. (It will be the same for both arrays.)

I: array index.

SutableIndex: index of the element that belongs in position I of array Date after the array is sorted.

**Figure 10.5**  
Data flow diagram  
for SortLists.

cedure StringExchange, it is routine to generate the code for the procedure SortLists as well as the rest of the program.

## Self-Test Exercises

1. What is the output of the following program?

```

program Exercisel(input, output);
const MaxLength = 5;
var A: array[1..MaxLength] of char; I : integer;
begin{Program}
  A[1] := 'a';
  for I := 2 to MaxLength do
    A[I] := 'b';
  writeln(A)
end. {Program}

```

(Self-Test Exercises continued on page 386)

**Program**

```

program HistoryTable;
{Reads in a list of composers and their years of birth.
Outputs the list sorted by birth date.
This program works in TURBO Pascal, but not in standard Pascal.}
const MaxLength = 20;
      MaxI = 10;
type NameString = string[MaxLength];
      Index = 1 .. MaxI;
      NameList = array[Index] of NameString;
      DateList = array[Index] of integer;
var Name: NameList;
    Date: DateList;
    Last: Index;

procedure ReadLists(var Name: NameList;
                   var Date: DateList; var Last: Index);
{Reads in a list of composers and their birth dates.
Postcondition: Date[I] is the year of birth of Name[I],
for all I <= Last; Last is the last index used.}
var I: integer;
    Ans: char;
begin{ReadLists}
  I := 0;
  repeat
    I := I + 1;
    writeln('Name? ');
    readln(Name[I]);
    writeln('Year of birth? ');
    readln(Date[I]);
    writeln('More? (y/n) ');
    readln(Ans)
  until (Ans = 'n') or (Ans = 'N');
  Last := I
end; {ReadLists}

procedure WriteLists (var Name: NameList;
                     var Date: DateList; Last: Index);
{Outputs the names and dates in the two arrays with
Name[I] and Date[I] paired. Uses indexes up through Last.
Precondition: array elements 1 through Last are defined.}
const Blank = ' ';
var I: integer;

```

**Figure 10.6**  
History table  
program.



```

begin{WriteLists}
  for I := 1 to Last do
    begin{One Composer}
      write(Date[I]:4, Blank);
      writeln(Name[I])
    end {One Composer}
  end; {WriteLists}

function Imin(var A: DateList; Start, Last: Index): Index;
{Returns the index I such that A[I] is the smallest of the values: A[Start], A[Start + 1], . . . . A[Last].}
    ...
    <The Rest of the declaration is given in Figure 9.16.>
    ...

procedure Exchange(var X, Y: integer);
{Interchanges the values of X and Y.}
var Temp: integer;
begin{Exchange}
  Temp := X;
  X := Y;
  Y := Temp
end; {Exchange}

procedure StringExchange(var X, Y: NameString);
{Interchanges the values of X and Y.}
var Temp: NameString;
begin{StringExchange}
  Temp := X;
  X := Y;
  Y := Temp
end; {StringExchange}

procedure SortLists(var Name: NameList;
                    var Date: DateList; Last: Index);
{Sorts the array Date into increasing order and moves the names to
follow the dates. Postcondition: The array elements of Date have been
rearranged so that Date[1] <= Date[2] <= . . . <= Date[Last];
if Date[I] was moved to Date[J], then Name[I] was moved to Name[J].}
var I, SuitableIndex: Index;
begin{SortLists}
  for I := 1 to Last - 1 do
    begin{Set Ith entries}
      SuitableIndex := Imin(Date, I, Last);
      Exchange(Date[I], Date[SuitableIndex]);
      StringExchange(Name[I], Name[SuitableIndex])
      {Date[1]<=Date[2]<= . . . <=Date[I] are the I smallest of the original
      array elements; the remaining elements are in the remaining positions.}
    end {Set Ith entries}
  end; {SortLists}

```

**Figure 10.6**  
(continued)

```
begin{Program}  
  writeln('Enter the composers and years of birth:');  
  ReadLists(Name, Date, Last);  
  SortLists(Name, Date, Last);  
  writeln('The composers in order of birth date:');  
  WriteLists(Name, Date, Last)  
end. {Program}
```

#### Sample Dialogue

Enter the composers and years of birth:

Name?

**Beethoven**

Year of birth?

**1770**

More? (y/n)

**yes**

Name?

**J. S. Bach**

Year of birth?

**1685**

More? (y/n)

**y**

Name?

**Mozart**

Year of birth?

**1756**

More? (y/n)

**y**

Name?

**Bloch**

Year of birth?

**1880**

More? (y/n)

**no**

The composers in order of birth date:

1685 J. S. Bach

1756 Mozart

1770 Beethoven

1880 Bloch

**Figure 10.6**  
**(continued)**

---

(Self-Test Exercises continued from page 383)

2. The procedure `StringWriteLn` in Figure 10.1 always ends by going to the next line. Hence, it cannot be used to write a string followed by a number all on one line. Write a procedure called `StringWrite` (no “`ln`”) that writes but does not advance to the next line. Thus, the output of the following will all appear on one line:

---

```
StringWrite (Name[I]);
write(' is numbered ', I)
```

3. Give suitable type declarations for the following data: the family records for the children in a large family. The family is so large that the children are numbered 1 through 17. The records are made up of each child's name, his or her year of birth, and whether or not the child has been vaccinated against the flu.

---

A man should keep his little brain attic stocked  
with all the furniture that he is likely to use,  
and the rest he can put away in the lumber-room of  
his library, where he can get it if he wants to.

*Sir Arthur Conan Doyle*  
(*Sherlock Holmes*), *Five Orange Pits*

---



---

## The Notion of a Data Structure

The notion of a data structure and the notion of a data type are intimately intertwined, and so we will preface our discussion of data structures with a review of the notion of a data type. Recall that a data type is a collection of values together with the operations that are provided for those values. For example, the values of the type `integer` are 0, 1, -1, 2, and so forth. The operations include addition, subtraction, multiplication, *mod*, *div*, and the comparison relations, such as  $<$  and  $=$ . The data type `real` has a different set of values and a different set of operations, although some operations, like addition, can apply to both of these data types. Certain operations relate two or more data types. For example, the function `sqrt`, when applied to a value of type `integer`, produces a value of type `real`. The function `round`, when applied to a value of type `real`, produces a value of type `integer`.

Yet another data type is the array type:

```
type List = array[0 . . 4] of real;
```

The values of this data type are lists of five numbers indexed by the numbers zero through four. The operations for this data type do not combine two values of this type but instead combine a value of this array type with a value of the subrange type `0 . . 4` to produce a value of type `real`. If *A* is of type `List`, for example, then *A* combined with 0 yields the indexed variable *A*[0]. In this case, the operation is `[ ]`. The syntax is a bit unorthodox for an operation, and the result, strictly speaking, is not a value but a variable. However, it is an operation, and it does produce a value of type `real`, namely the value of the indexed variable *A*[0].

Like the data types `integer`, `real`, and `char`, the type `List` is a data type with values and operations. However, it clearly is a different sort of type. It is what we call a *structured data type*. It is called “structured” because it has a structure, namely

---



a list of numbers, that can be decomposed into parts, namely the individual real numbers, by means of operations, namely using `[ ]` to produce indexed variables like `A[0]`.

A data structure is almost the same thing as a structured data type. Every structured data type is a data structure, and given any data structure, we could redesign the Pascal language to include it as a data type. A *data structure* is a way of structuring data. It organizes the data into composite items and provides operations for manipulating the items. A data structure is like a structured data type in all ways but one: It need not be a declared data type that is named in the type declaration part of the program. For example, consider the parallel arrays that we used in the history table case study. Each separate array type is a data type, but the data structure consists of those two arrays taken together as a unit. There is no data type called “parallel arrays,” but there is a data structure called “parallel arrays.” Data types are in the declaration section of the program. Data structures are in the mind of the programmer, and might or might not also occur in the declaration section of the program.

---

## Data Abstraction

We have already discussed the concept of procedural abstraction. It consists of forgetting the inessential details of a procedure. Wise use of procedural abstraction means that once we have finished writing and testing a procedure, we can forget how the procedure works and concern ourselves only with what it does. Once we have written (or someone else has written) the procedure `StringReadLn` in Figure 10.1, we need not constantly return to the code to see how it works. We need only remember that it fills an array of type `CharString` with characters read from the keyboard and that it pads the string with blanks to make it 20 characters long. There is no need to remember whether it uses a *for* loop or a *while* loop or even whether it uses any loop at all. This forgetting frees our minds of inessential details and so frees our mental energy for the larger details of program writing.

*Data abstraction* is a similar kind of judicious forgetfulness, but in this case it is applied to data structures rather than to procedures. In order to realize a data structure within Pascal or any other programming language, we may need to specify some details which, while necessary in the particular language, are not a necessary part of our intuitive thinking about the data structure. For example, the type `CharString` discussed in the first few sections of this chapter was designed to hold strings of 20 or fewer characters. A natural way to do this is with an array. This required that we specify an index type. We chose to specify `1 .. 20`. We might just as well have chosen `0 .. 19`. Some authorities would argue vigorously that that would have been a better choice for an index type. Nobody is likely to argue for `100 .. 119` as the index type, but it would have been adequate. Since the index type matters so little, it would seem safe to just forget about it. This is exactly what data abstraction is all about: forgetting the inessential details of a data structure. The productive way to view the type `CharString` is as strings of 20 or fewer characters and not as an array type with some arbitrary index

---

type. You should think about designing algorithms that manipulate strings. Do not waste mental energy memorizing the particular numbers used to index the characters.

Data abstraction and procedural abstraction go hand in hand. Once a data structure like the type `CharString` has been designed, and once the procedures for the basic operations like `StringReadln` and `StringWriteln` have been written, we can forget the inessential details of both the data structure and the procedures. At that point we are reading and writing strings of characters, not arrays of characters, and we are doing it with procedures, not with *for* loops.

---

## TURBO Pascal

---

### Clearing the Screen

TURBO Pascal has a predefined procedure called `clrscr` which clears the screen. In order to use this procedure, your program must contain the following line immediately after the program heading:

```
uses crt;
```

`crt` is a collection of additional predefined procedures including the procedure `clrscr`. The use of the procedure `clrscr` is illustrated in the next case study.

---

## TURBO Pascal Case Study

---

### Automated Drill

#### Problem Definition

Earlier in this chapter, we designed a program to produce a history table listing composers and their years of birth. Now we want to produce a related automated drill program. The program will accept the same input and build the same parallel arrays, but instead of outputting a table, it will drill the user to see whether the user knows the birth dates of the composers. After building the parallel arrays, the program will clear the screen. It will then ask the user to input a composer and birth date and will check to see whether or not the birth date is correct. It will repeat this drill as often as the user desires.

#### Discussion

An outline for the program is the following:

1. Input data and fill the parallel arrays.
  2. Clear the screen.
-

3. Do the following as often as the user wants:
  - a. Read a composer's name and candidate year for birth date.
  - b. Search the list of names to find the composer.
  - c. Output an appropriate response.

This program can be written by adapting procedures that we have already designed for other tasks. The input and filling of the parallel arrays can be done as in the history table program. The TURBO Pascal procedure `clrscr` can be used to clear the screen. The input of a composer's name and candidate birth year is standard, as is the output message. Only the search algorithm seems to present anything new, but even it can be adapted from previous algorithms.

#### ALGORITHM

*serial  
search*

The search algorithm will be given an array of names. We will again use the array type `NameList`. The type definition is reproduced below:

```
type NameString = string[MaxLength];
   Index = 1..MaxI;
   ExtendedIndex = 0..MaxI;
   NameList = array[Index] of NameString;
```

The search will need to return the index of the name if the name is found and will need to indicate when the name is not on the list. Thus, we are led to the following function heading:

```
function Search(Pattern: NameString;
               var Name: NameList; Last: Index): ExtendedIndex;
{Returns the index I such that Name[I] = Pattern provided there is
such an I in the range 1 through Last; otherwise returns 0.}
```

In Figure 9.10 we used a serial search algorithm to search an array of integers. That algorithm used no special properties of integers, and so it can be adapted to search any list, including a list of strings. The following is a version of this serial search algorithm that tests each element of the array `Name` to see whether `Pattern` is present:

```
I := 1;
while (I <= Last) and Pattern not found do
  begin{while}
    if Pattern = Name[I] then
      Pattern is found
    else
      I := I + 1
  end; {while}

if Pattern is found then
  return I
else
  return 0
```

The complete program is given in Figure 10.7.



**Program**

```

program DrillDates;
{Reads a list of composers and their years of birth,
then drills the user on years of birth.
This program works in TURBO Pascal, but not in standard Pascal.}
uses crt;
const MaxLength = 20;
      MaxI = 10;
type NameString = string[MaxLength];
      Index = 1 .. MaxI;
      ExtendedIndex = 0 .. MaxI;
      NameList = array[Index] of NameString;
      DateList = array[Index] of integer;
var Pattern: NameString;
    Name: NameList;
    Date: DateList;
    Last: Index;
    Ans: char;
    Year: integer;
    Outcome: ExtendedIndex;

procedure ReadLists(var Name: NameList;
                    var Date: DateList; var Last: Index);
{Reads in a list of composers and their birth dates.
Postcondition: Date[I] is the year of birth of Name[I],
for all I <= Last; Last is the last index used.}
    . . .
    <The rest of the declaration is given in Figure 10.6>
    . . .

function Search(Pattern: NameString;
                var Name: NameList; Last: Index): ExtendedIndex;
{Returns the index I such that Name[I] = Pattern provided there is
such an I in the range 1 through Last; otherwise, returns 0.}
var I: integer;
    Found: boolean;
begin{Search}
    Found := false; {not found so far.}
    I := 1;
    while (I <= Last) and (not Found) do
        begin{while}
            if Pattern = Name[I] then
                Found := true
            else
                I := I + 1
        end; {while}
end;

```

**Figure 10.7**  
Automated drill  
program.

```

    if Found then
        Search := I
    else
        Search := 0
end; {Search}

begin{Program}
    writeln('Enter the composers and years of birth: ');
    ReadLists(Name, Date, Last);

    clrscr;
    writeln('Do you want to test your memory of dates? (y/n) ');
    readln(Ans);
    while (Ans = 'y') or (Ans = 'Y') do
        begin{while}
            writeln('Enter composer's name: ');
            readln(Pattern);
            writeln('Year of birth? ');
            readln(Year);
            Outcome := Search(Pattern, Name, Last);
            {Outcome is index of composer; Outcome = 0 if not on the list.}
            if Outcome <= 0 then
                writeln('Sorry, no record of that composer. ')
            else if Year = Date[Outcome] then
                writeln('You're right! ')
            else {Year is wrong and composer was found.}
                begin{give correct year}
                    writeln('Sorry, you are wrong. ');
                    writeln(Name[Outcome]);
                    writeln('was born in ', Date[Outcome])
                end; {give correct year}
                writeln('Again? (y/n) ');
                readln(Ans)
            end; {while}
        writeln('End of drill. ')
    end. {Program}

```

### Sample Dialogue

```

Enter the composers and years of birth:
Name?
Bloch
Year of birth?
1880
More? (y/n)
y

```

**Figure 10.7**  
(continued)

Name?

**J. S. Bach**

Year of birth?

**1685**

More? (y/n)

**y**

Name?

**Beethoven**

Year of birth?

**1770**

More? (y/n)

**no**

Screen is cleared.

Do you want to test your memory of dates? (y/n)

**yes**

Enter composer's name:

**Beethoven**

Year of birth?

**1776**

Sorry, you are wrong.

Beethoven

was born in 1770

Again? (y/n)

**y**

Enter composer's name:

**Mozart**

Year of birth?

**1756**

Sorry, no record of that composer.

Again? (y/n)

**y**

Enter composer's name:

**J. S. Bach**

Year of birth?

**1685**

You're right!

Again? (y/n)

**n**

End of drill.

**Figure 10.7**  
**(continued)**



## Adapting a Known Algorithm

The previous case study illustrates one very important programming technique. Problems are very often variations on previously solved problems. Before trying to construct an algorithm from scratch, it always pays to see if the problem is similar to one that you solved previously. If the previously written program is well documented, this will be easy to do. It may even be possible to use entire procedures from previously written, well-documented programs.

---

## Standard Pascal Case Study

---

### Pattern Matching

In this section we will design a pattern-matching algorithm for strings represented as arrays of characters and will implement the algorithm as a boolean-valued function. This will be a portable function designed in standard Pascal. It will work in either standard Pascal or TURBO Pascal or, for that matter, in virtually any version of Pascal. Strings will be stored as arrays of characters of the following type, which was introduced in the first section of this chapter:

```
type CharString = array[1..MaxLength] of char;
```

### Problem Definition

We will design a boolean-valued function `Subpattern` which has two parameters called `Pattern` and `Target`, both of type `CharString`. The function will tell us whether or not the string contained in `Pattern` is a substring of the string contained in `Target`. For example, if `Pattern` contains "Bach" and `Target` contains "J.S. Bach," then the function should return `true`. On the other hand, if `Pattern` contains "Berg" and `Target` contains "J.S. Bach," then the function should return `false`.

Both the pattern, like "Bach," and the string that it is compared to, like "J.S. Bach," will be stored in arrays of characters of the type `CharString`. The string characters will be stored in the arrays starting at indexed variable 1, and any unused indexed variables at the end of the arrays will be filled with the blank symbol, which is what would happen if the strings were read in by the procedure `StringReadln` in Figure 10.1. Since the strings are typically of length less than `MaxLength`, there is a problem with trailing blanks. The string "Bach" will be stored as "Bach," followed by sixteen blanks. Our pattern-matching function will need to somehow disregard any trailing blanks. The function we want can be described by the following heading:

```
function Subpattern(var Pattern, Target: CharString): boolean;  
{Returns true if the string in Pattern occurs as a substring of the string  
in Target; otherwise returns false. Disregards trailing blanks in Pattern.}
```

---

## Discussion

Suppose that the array `Pattern` contains the string “Bach” in indexed variables `Pattern[1]` through `Pattern[4]`, followed by trailing blanks. We want to know whether “Bach” occurs anywhere in the array `Target`. One way to approach the problem is to assume that the arrays are strips of paper containing the strings and to visualize how we might compare the two strings using the strips of paper. This approach to the problem is diagrammed in Figure 10.8. For each position in the target array, we compare the pattern to the substring beginning at that position. Our approach is nothing but a more involved version of the serial search algorithm we used earlier, and it translates to the following algorithm outline:

Perform the following until either a match is found or

`I` is larger than the last possible start index for a pattern match:

*begin*{*Testing at Target[I]*}

    compare

`Pattern[1]` through `Pattern[<length of the pattern>]` to  
        `Target[I]`, `Target[I + 1]`, `Target[I + 2]`, etc.

*if* they match *then*

        the pattern is found

*else*

`I := I + 1` {*Move “the strip of paper” over one square.*}

*end* {*Testing at Target[I]*}

if no match is found, then `Target` does not contain the pattern.

If the pattern string is “Bach,” our algorithm needs to know that there are four letters in “Bach.” It also needs to be able to compare “Bach” to any four successive elements of the target array. We will need to design a function or procedure for each of these two subtasks: one to find the length of the pattern and another to match the pattern array to a portion of the target array.

The length of the pattern will be computed by the function described in the following heading:

*function* `Length`(`var A: CharString`): `integer`;  
    {*Returns the length of the string in A, not counting blanks at the end.*  
    *Precondition: All indexed variables of A have a value; A is not all blanks.*}

The most obvious approach is to count the characters starting at the beginning of the string and to stop when we encounter a blank. However, this approach will not work for strings that contain blanks, such as “Vaughan Williams.” To accommodate such strings, we will instead count backward from the end of the array until we find a non-blank character. We count by decrementing a variable `Count` as follows:

```
Count := MaxLength;
while A[Count] = Blank do
    Count := Count - 1;
Length := Count
```

The complete function declaration for `Length` can be found in Figure 10.9.

## ALGORITHM

## Length function



**Figure 10.8**  
**Technique for**  
**finding a pattern**

### Match function

We now turn to the task of comparing the pattern to one portion of the array `Target`. In other words, we want to design a module with some parameters that can perform any one of the various comparisons illustrated in Figure 10.8. For our parameters we will want names that are more meaningful than `I` but shorter than `Length(Pattern)`. Figure 10.10(a) illustrates our task and presents our naming conventions. `EndP` is the value `Length(Pattern)`. `StartT` is the index of `Target` at which the comparison begins. If `StartT` has a value of 3, as in Figure 10.10(a), the comparison will fail. If it has a value of 5, and the other values are as



```

const MaxLength = 20;
type CharString = array[1..MaxLength] of char;

function Length(var A: CharString): integer;
{Returns the length of the string in A not counting blanks at the end.
Precondition: All indexed variables of A have a value; A is not all blanks.}
const Blank = ' ';
var Count: integer;
begin{Length}
  Count := MaxLength;
  while A[Count] = Blank do
    Count := Count - 1;
  Length := Count
end; {Length}

```

**Figure 10.9**  
**The function**  
**Length.**

shown in Figure 10.10(a), the comparison will succeed. Our module will compare the following two lists of characters:

```

Pattern[1], Pattern[2], Pattern[3], ..., Pattern[EndP]
Target[StartT], Target[StartT + 1], Target[StartT + 2], etc.

```

A natural way to do this is with a boolean-valued function that behaves as indicated below:

```

function Match(var Pattern, Target: CharString;
               EndP, StartT: integer): boolean;
{Returns true if Pattern[1] through Pattern[EndP] matches the
characters in Target starting with Target[StartT]; otherwise, returns false.}

```

We will be applying the function Match only when the index position StartT is small enough to allow a match. For example, we will not try to compare a pattern of length four to the last three or fewer symbols in the target array. To ensure that we do not lose track of this assumption, we will make it an explicitly stated precondition:

*Precondition: The array Target contains at least EndP elements in index positions StartT through the end of the array.*

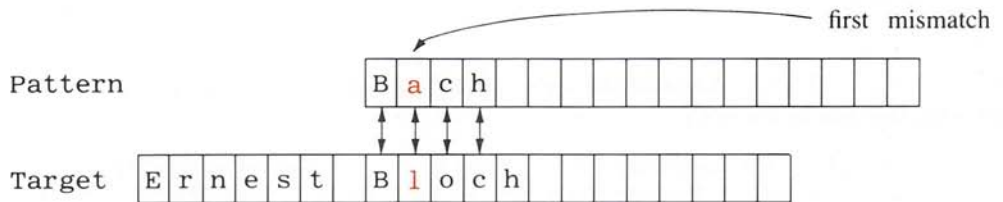
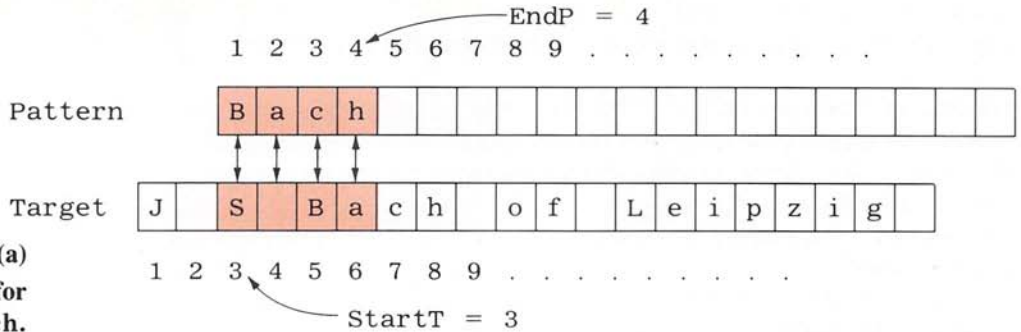
*precondition*

We will use a technique known as “guilty until proven innocent.” We assume that the pattern does match (“is guilty”) unless we can find two corresponding characters that do not match (and so prove that the pattern does not match the target). The technique is illustrated in Figure 10.10(b) and can be implemented with the following algorithm to compute the value returned by Match:

```

Assume that the pattern in Pattern does match the
symbols in Target[StartT], Target[StartT + 1], etc.
unless a mismatch of corresponding symbols is discovered.
I := 1;
Repeat the following until a mismatch is found
or the end of the pattern in Pattern is reached:

```



```

begin{Check one corresponding pair}
  if Pattern[I] <> Target[<corresponding index>] then
    a mismatch was found
  else
    I := I + 1
end; {Check one corresponding pair}

```

Using a boolean variable, MatchSoFar, we can easily realize the above algorithm as the following *while* loop:

```

MatchSoFar := true;
I := 1;
while MatchSoFar and (I <= EndP) do
  begin{Check one corresponding pair}
    if Pattern[I] <> Target[<corresponding index>] then
      MatchSoFar := false
    else
      I := I + 1
    {MatchSoFar is true unless there is a mismatch before position I of Pattern.}
  end; {Check one corresponding pair}

```

After the loop terminates, the value of MatchSoFar is returned as the value of the function Match. All that we need to complete the definition of Match is to obtain a Pascal expression for <corresponding index>.

To find the correct expression for the corresponding index, try some concrete examples. If  $I$  is equal to 1, then the corresponding index is  $\text{StartT}$ . If  $I$  is equal to 2, then the corresponding index is  $\text{StartT} + 1$ . This process reveals the pattern. The correct expression is  $\text{StartT} + I - 1$ .

It is now routine to convert our algorithm to a Pascal function declaration. The complete function declaration for `Match` can be found in Figure 10.11.

We are now ready to piece together the complete pattern-matching algorithm for the function `Subpattern`. The basic algorithm uses a technique which is a slight variant on the “guilty until proven innocent” technique. This variant is called the “innocent until proven guilty” technique. We assume that the pattern is not there (“innocent”), unless we find it and thereby “prove” that it is there (“guilty”). The algorithm can be realized using a boolean variable `Found` which is tentatively set equal to `false` to denote the absence of a match (“not guilty”) and only changed to `true` (“guilty”) if a match is discovered. The algorithm is

```
Found := false; {so far}
I := 1;
while (not Found) and (I <= MaxLength - EndP + 1) do
  if Match(Pattern, Target, EndP, I) then
    Found := true
  else
    I := I + 1
```

The final value of `Found` is the value returned by the function `Subpattern`, as shown in Figure 10.12.

You should always check that a loop is not executed one too few or one too many times. In designing the above code, you might be tempted to use the following as the last value of the variable `I` in the `while` loop:

```
MaxLength - EndP
```

However, a careful check will reveal that this expression would stop the loop one iteration too early. This is easiest to see if you consider some concrete values. Recall that `MaxLength` is 20, and suppose that the length of the pattern is 4. The value of the above expression (without the  $+1$ ) is 16. Yet if we try 17 we will see that it is possible to fit in a pattern of four letters starting with index variable 17:

```
Target[17] = 'B' Target[18] = 'a'
                Target[19] = 'c' Target[20] = 'h'
```

Hence, we see that in fact we need to include the  $+1$  as shown in our `while` loop; otherwise, we will miss one possible location for the pattern.

*check boundary  
values*

---

## Design by Concrete Example

When you are designing a program or procedure, it is often more fruitful to think in terms of a concrete example rather than an abstract characterization. In the previous example, it helped to think of locating the pattern “Bach” in the target string “J S Bach”  
(continued, page 404)

---



**Program**

```

program TestMatch(input, output);
{Tests the function Match.}
const MaxLength = 20;
type CharString = array[1..MaxLength] of char;
var Pattern, Target: CharString;
    StartT, I: 1..MaxLength;
    Ans: char;

procedure StringReadln(var A: CharString);
{Fills the array A with one line of input characters and fills any extra positions with blanks.
If there are too many characters it discards all characters after the first MaxLength characters.}

```

<The rest of the declaration is given in Figure 10.1>

```

function Length(var A: CharString): integer;
{Returns the length of the string in A not counting blanks at the end.
Precondition: All indexed variables of A have a value; A is not all blanks.}

```

<The rest of the declaration is given in Figure 10.9>

```

function Match(var Pattern, Target: CharString;
               EndP, StartT: integer): boolean;
{Returns true if Pattern[1] through Pattern[EndP] matches the
characters in Target starting with Target[StartT]; otherwise, returns false.
Precondition: The array Target contains at least EndP elements in
index positions StartT through the end of the array.}
var I: integer;
    MatchSoFar: boolean;
begin {Match}
    MatchSoFar := true;
    I := 1;
    while MatchSoFar and (I <= EndP) do
        begin {Check one corresponding pair}
            if Pattern[I] <> Target[StartT + I - 1] then
                MatchSoFar := false
            else
                I := I + 1
            {MatchSoFar is true unless there is a mismatch before position I of Pattern.}
        end; {Check one corresponding pair}
    Match := MatchSoFar
end; {Match}

```

**Figure 10.11**  
**Program**  
**containing the**  
**function Match.**

```

begin{Program}
  repeat
    writeln('Enter a target string and press return. ');
    for I := 1 to MaxLength do
      write(I mod 10:1); {To make it easy to count.}
    writeln;
    StringReadln(Target);
    writeln('Enter a pattern string and press return. ');
    StringReadln(Pattern);
    writeln('Enter the index of where you want');
    writeln('to try matching the target. ');
    readln(StartT);
    if Match(Pattern, Target, Length(Pattern), StartT) then
      writeln('The pattern was FOUND at index ', StartT)
    else
      writeln('The pattern was NOT FOUND at index ', StartT);
    writeln('More? (y/n) ');
    readln(Ans)
  until (Ans = 'n') or (Ans = 'N');
  writeln('End of test. ')
end. {Program}

```

### Sample Dialogue

```

Enter a target string and press return.
12345678901234567890
J S Bach of Leipzig
Enter a pattern string and press return.
Bach
Enter the index of where you want
to try matching the target.
3
The pattern was NOT FOUND at index    3
More? (y/n)
y
Enter a target string and press return.
12345678901234567890
J S Bach of Leipzig
Enter a pattern string and press return.
Bach
Enter the index of where you want
to try matching the target.
5
The pattern was FOUND at index    5
More? (y/n)
n
End of test.

```

**Figure 10.11**  
(continued)

**Program**

```

program TestSubpattern(input, output);
{Tests the procedure Subpattern.}
const MaxLength = 20;
type CharString = array[1..MaxLength] of char;
var Pattern, Target: CharString;
    Ans: char;

procedure StringReadln(var A: CharString);
{Fills the array A with one line of input characters and fills any extra positions with blanks.
If there are too many characters it discards all characters after the first MaxLength characters.}

```

.<The rest of the declaration is given in Figure 10.1>  
 . . .

```

function Length(var A: CharString): integer;
{Returns the length of the string in A not counting blanks at the end.
Precondition: All indexed variables of A have a value; A is not all blanks.}

```

.<The rest of the declaration is given in Figure 10.9>  
 . . .

```

function Match(var Pattern, Target: CharString;
               EndP, StartT: integer): boolean;
{Returns true if Pattern[1] through Pattern[EndP] matches the
characters in Target starting with Target[StartT]; otherwise, returns false.
Precondition: The array Target contains at least EndP elements in
index positions StartT through the end of the array.}

```

.<The rest of the declaration is given in Figure 10.11>  
 . . .

```

function Subpattern(var Pattern, Target: CharString): boolean;
{Returns true if the string in Pattern occurs as a substring of the string
in Target; otherwise, it returns false. Disregards trailing blanks in Pattern.
Calls the functions Length and Match. Precondition: Pattern is not all blanks.}
var EndP: integer;
    Found: boolean;
    I: integer;

```

**Figure 10.12**  
**Finding a pattern.**



```

begin{Subpattern}
  EndP := Length(Pattern);
  Found := false; {so far}
  I := 1;
  while (not Found) and (I <= MaxLength - EndP + 1) do
    if Match(Pattern, Target, EndP, I) then
      Found := true
    else
      I := I + 1;
  Subpattern := Found
end; {Subpattern}

begin{Program}
  repeat
    writeln('Enter a target string and press return. ');
    StringReadln(Target);
    writeln('Enter a pattern string and press return. ');
    StringReadln(Pattern);
    if Subpattern(Pattern, Target) then
      writeln('The pattern was FOUND. ')
    else
      writeln('The pattern was NOT FOUND. ');
    writeln('More? (y/n) ');
    readln(Ans)
  until (Ans = 'n') or (Ans = 'N');
  writeln('End of test. ')
end. {Program}

```

### Sample Dialogue

```

Enter a target string and press return.
Ernest Bloch
Enter a pattern string and press return.
Bach
The pattern was NOT FOUND.
More? (y/n)
y
Enter a target string and press return.
J S Bach of Leipzig
Enter a pattern string and press return.
Bach
The pattern was FOUND.
More? (y/n)
n
End of test.

```

**Figure 10.12**  
(continued)

of Leipzig.” This allowed us to visualize the problem and to try out our ideas immediately. Once we knew how to find “Bach” in “J S Bach of Leipzig,” we could then replace “Bach” with an arbitrary array *Pattern* and “J S Bach of Leipzig” with another arbitrary array *Target*.

When using this technique, you should consider one concrete case to represent each of the major possibilities. In this case, either the pattern is found or it is not. Hence, we also needed to look at a value for *Target* that does not contain the pattern “Bach.” For this we used the target string “Ernest Bloch.” Once you have an algorithm that works for a few special cases, it is not guaranteed that you have the correct algorithm, but you do have some inspiration. Once you have some inspiration and a candidate algorithm, you must determine whether the algorithm works for all cases, and if it does not, you must modify it so that it does.

The technique of using concrete data is particularly fruitful when you are determining the exact form of an arithmetic expression. In designing the algorithm for the procedure *Subpattern*, we needed to know the last index of *Target* that allowed sufficient room to hold a pattern of size  $\text{Length}(\text{Pattern})$ . Clearly, the number is approximately

$$\text{MaxLength} - \text{Length}(\text{Pattern})$$

but such expressions can often be off by one in either direction. When we considered the concrete examples of 4 for the pattern length and 20 for the *MaxLength*, we immediately saw that we needed to add one to obtain the correct expression. When deciding the exact form of an arithmetic expression, you will find that nothing clarifies your thinking as well as a concrete example.

---

## Standard Pascal-Packed Arrays (Optional)

There is a special class of arrays in Pascal that are supposed to be implemented so as to save storage. These are the *packed* arrays. They are not absolutely necessary, but on some systems they do save storage.

The Pascal language does not specify exactly how an array must be implemented. However, the language does specify that there will be two different implementations of arrays. One implementation is supposed to make the program run faster but may use more storage. That form consists of the basic array types we have been discussing. The other implementation is supposed to save storage but may make the program run slower.

These storage-efficient arrays are called “packed arrays” and are declared just like ordinary arrays except that the identifier *packed* is inserted before the word *array*. For example,

```
type SqueezeNumbers = packed array[0 . . 100] of integer;
```

The word *packed* instructs the compiler to use memory more efficiently. A typical scenario is as follows. Without the word *packed*, the compiler will reserve one

---

memory location for each indexed variable. When the word *packed* is included, the compiler will instruct the computer to place ("pack") as many array elements as possible into one memory location.

Packed arrays are used in basically the same way as ordinary arrays. For example, if a packed array A is declared by

```
var A: SqueezeNumbers;
```

then it can be used in the same way as an ordinary array of integers, except in the few special cases discussed below.

Packed arrays save storage and so, all other things being equal, it should be preferable to use packed arrays rather than ordinary arrays. However, "all other things" are usually far from equal. Packed arrays have a number of disadvantages. Hence, they are usually used only when storage efficiency is a major issue. One disadvantage of packed arrays is that they tend to make the program run slower.

One of the biggest disadvantages of packed arrays has to do with using them as parameters to procedures. On some systems an indexed variable of a packed array cannot be passed as a variable parameter to a procedure. Given a procedure heading of

```
procedure Exchange (var X, Y: integer);
```

and a variable A, declared as above, some systems will not allow the following procedure call:

```
Exchange (A[1], A[2])
```

This may have repercussions beyond user-defined procedures. On some systems the same prohibition applies to the standard procedures `read` and `readln`. On those systems data must be read into some other type of variable and then transferred to the packed array.

Other problems with packed arrays arise from the fact that packed arrays are considered to be a different type than the corresponding ordinary array type. Given the declarations

```
var L: array[1..80] of char;
    PL: packed array[1..80] of char;
```

the assignment `L := PL` produces a type conflict and is not allowed. Similarly, if a packed array is to be the actual parameter to a procedure, the formal parameter must also be a packed type.

*restrictions  
on packed  
arrays*

---

## Standard Pascal-Packed Arrays of Characters (Optional)

There are occasions when storage efficiency is very important. Hence, despite all their disadvantages, packed arrays can sometimes be very useful. The storage savings resulting from using packed arrays may be particularly large in the case of arrays of characters. Usually several characters can be packed into one memory location. Hence, if

---



storage is a major consideration, the type `CharString`, discussed earlier in this chapter, could be redefined to be a packed array of characters:

```
const MaxLength = 20;
type CharString = packed array[1..MaxLength] of char;
var Line: CharString;
```

Although we are using the same identifier `CharString` to name this type, it is a different type from the previously defined type `CharString`, and it has certain special properties.

*assignment  
of strings*

Unlike ordinary arrays, it is possible to fill a packed array of characters by using an assignment operator and a string constant. Packed arrays of characters are also allowed as arguments to the `write` and `writeln` statements. For example, with `Line` declared as in the previous paragraph, the following code is allowed:

```
Line := 'It is tight in here.';
writeln(Line)
```

The output produced by these two lines is

```
It is tight in here.
```

Some systems allow the use of a packed array of characters in a `read` or `readln` statement, as a way of reading a string constant into the array. However, many other systems do not allow the `read` and `readln` statements to be used in this way.

When filling a packed array with a string constant, remember that the length of the array is important: Any constant assigned to it must have exactly the same number of characters as the array has elements. To place the constant `'Loose fit'` into the array `Line` declared above, you must pad it with 11 spaces, like so:

```
Line := 'Loose fit                ';
```

Packed arrays of characters can be compared using the less-than relation `<`. The ordering is approximately alphabetical. However, the exact result of a comparison will vary from one implementation to another.

---

A penny saved is a penny earned.  
*Proverb*

---

Penny wise, pound foolish.  
*Robert Burton, Anatomy of Melancholy*

---

---

## Self-Test Exercises

4. How must the data type `CharString` and the procedures `StringReadln` and `StringWriteln` of Figure 10.1 be modified to accommodate strings of up to 40 characters?
5. (This exercise uses the optional material on packed arrays of characters.) Given the declaration

```
type Words = packed array[1..5] of char;  
var A: Words;
```

which of the following are allowed?

```
A := 'Stuff';  
write(A);  
writeln(A);  
read(A);  
A := 'Hi!'
```

---

## Multidimensional Arrays

An index often contains some sort of information about how the array elements are most naturally organized. For some situations, one index is not sufficient to organize the elements in an ideal fashion. As an example, suppose that we wish to hold a page of text in an array. We could number the characters of the text consecutively and use a larger array index range. If there are 100 characters per line and 30 lines per page, we could make it an array of characters indexed by the type `1..3000`. However, it would be much more convenient to use two different indexes, one for the line and one for the character on that line. In Pascal and most other high level programming languages, this is possible. It is possible, for example, to have an array called `P` that has two indexes, one for the line and one for the character in that line. Although Pascal does not insist on it, it is traditional to make the first index count the lines and the second index count the position in the line. The array declaration in this case would be written as follows:

*multiple  
indexes*

```
type OnePage = array[1..30, 1..100] of char;  
var P: OnePage;
```

With `P` declared in this way, the first character of the first line can be stored as the value of `P[1, 1]`; the second character on the first line can be stored as the value of `P[1, 2]`, and so forth. As another sample, the value of

```
P[5, 38]
```

---

is the thirty-eighth character on the fifth line. To write out the entire fifth line to the screen, the following will do:

```
for I := 1 to 100 do
    write(P[5, I]);
writeln
```

*two-dimensional  
example*

An array with more than one index is called a *multidimensional* array. The diagram in Figure 10.13 may help to explain this choice of terminology. The array A depicted there contains a list of average grades for a small class with four students numbered 1 through 4. The array can be thought of as a simple list. Such lists are one-dimensional objects. The array G shows more details of the class grading. It gives three quiz scores for each of the four students. This can be visualized as a two-dimensional arrangement, with one row for each student and one column for each quiz.

*sample  
declarations*

Multidimensional arrays are declared in the same way as the one-dimensional arrays we have been studying until now. The syntax for a multidimensional type definition is described in Figure 10.14. As is the case with one-dimensional arrays, the index types for multidimensional arrays must be ordinal or subrange types. The various index types need not be subranges of the same type. There may be any number of indexes, so

```
var A: array[1..4] of real;
```

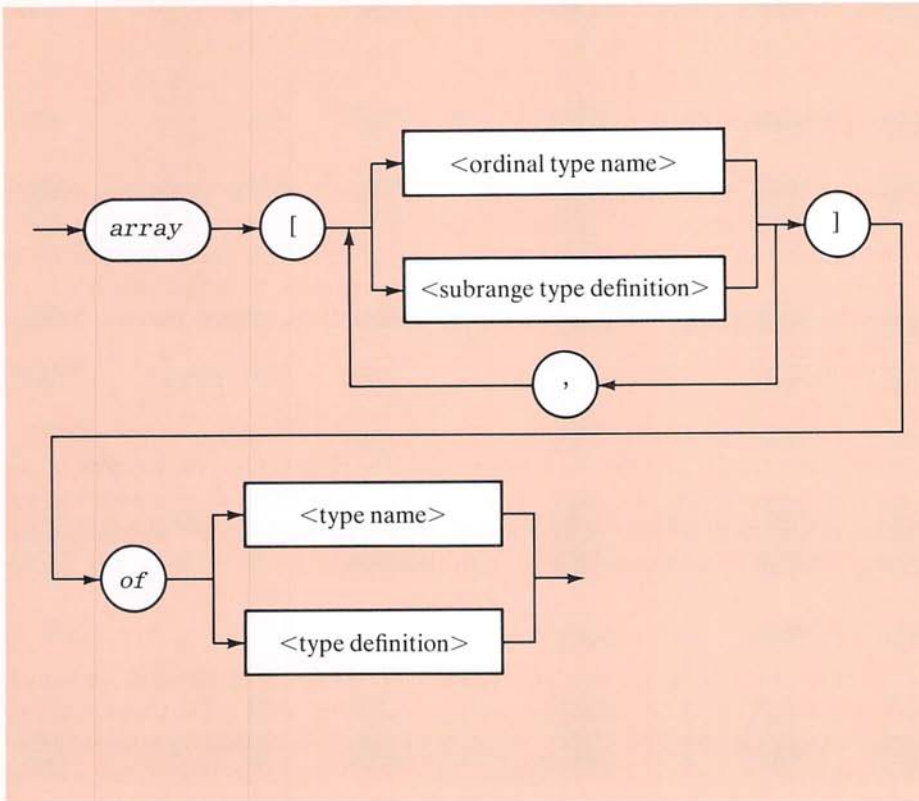
Layout		Sample Values	
	Average		Average
Student 1	A[1]	Student 1	10.0
Student 2	A[2]	Student 2	1.0
Student 3	A[3]	Student 3	7.7
Student 4	A[4]	Student 4	7.7

```
var G: array[1..4, 1..3] of 0..10;
```

Layout				Sample Values			
	Quiz 1	Quiz 2	Quiz 3		Quiz 1	Quiz 2	Quiz 3
Student 1	G[1, 1]	G[1, 2]	G[1, 3]	Student 1	10	10	10
Student 2	G[2, 1]	G[2, 2]	G[2, 3]	Student 2	2	0	1
Student 3	G[3, 1]	G[3, 2]	G[3, 3]	Student 3	8	6	9
Student 4	G[4, 1]	G[4, 2]	G[4, 3]	Student 4	8	5	10

**Figure 10.13**  
One- and two-  
dimensional  
arrays.





**Figure 10.14**  
**Syntax of array**  
**type definitions.**

long as there is at least one. (If there is just one, then it is a one-dimensional array.) As with one-dimensional arrays, the component type can be any Pascal type. Some sample type declarations for multidimensional arrays are

```
type Matrix = array[1..10, 1..5] of real;
Count = array[1..100, 'A'..'Z'] of integer;
Picture = array[0..4, 0..9] of char;
```

Arrays of the first type might hold real numbers for some scientific or engineering calculation. The type `Count` can be used in text processing. Suppose `C` is declared as

```
var C: Count;
```

The array C might then be used to count occurrences of each letter in a text of 100 lines. With the two-dimensional array, the program can easily keep a separate count for each line. The number of occurrences of, say, 'M' on, say, line 20 would be the value of C[20, 'M']. Arrays of type `Picture` might be used to hold patterns consisting of 5 lines of 10 characters each, which, when displayed on the screen, form a geometric pattern.

Aside from the fact that they may have more than one index, indexed variables of multidimensional arrays have the same properties as those of one-dimensional arrays.

sample applications

## Case Study

### Grading Program

To illustrate the use of multidimensional arrays, we will solve a simple class grading problem. The program we design will compute student averages and display the individual quiz grades and the averages for each student in the class. The program could be used by an instructor to obtain an overview of class grades. To give a perspective on how difficult each individual quiz was, the program will also display the class average for each quiz.

### Problem Definition

The quiz scores are to be read into the computer. The program will then display on the screen output consisting of each student's number followed by the student's average grade and a list of all the quiz scores for that student. The program will also display the average score of all the students for each individual quiz.

### Discussion

*data structure*

We need to devise some method for keeping track of each student's score. A natural way to do this is with a two-dimensional array, such as the array *G* illustrated in Figure 10.13. One index can be the student number; the other can be the quiz number. If the array is called *G*, then *G*[*SNum*, *QNum*] will contain the grade that student number *SNum* received on quiz number *QNum*. If *NumStudents* and *NumQuizzes* are constants equal to the number of students and the number of quizzes, and if each quiz is scored on a basis of 0 to 10 points, then the declaration for the two-dimensional array *G* can be as follows:

```
type Score = 0 . . 10;
  StudentIndex = 1 . . NumStudents;
  QuizIndex = 1 . . NumQuizzes;
  GradeArray =
    array[StudentIndex, QuizIndex] of Score;
  StuAveArray = array[StudentIndex] of real;
  QuizAveArray = array[QuizIndex] of real;
var G: GradeArray;
    SA: StuAveArray;
    QA: QuizAveArray;
```

Figure 10.13 illustrates one possible set of values for *G*. There and in our program, *NumStudents* is set to 4 and *NumQuizzes* is set to 3.

We will use two one-dimensional arrays to hold the averages. The array *SA* will hold the averages of each student, and the array *QA* will hold the class average for each quiz. The relation between these two arrays and the array *G* is illustrated in Figure 10.15. The arrays *SA* and *QA* are not absolutely necessary for this simple problem, but

they would be needed if we expanded the program to do more complicated tasks, such as displaying the student averages in sorted order or showing how much each score differs from the class average.

The task to be solved by this program can be decomposed into four subtasks:

*ALGORITHM*

1. Read in the quiz grades.
2. Compute the average for each quiz.
3. Compute the average for each student.
4. Display the grades and the averages.

Each task is accomplished by a separate procedure. The complete program is shown in Figure 10.16.

As illustrated in the procedure *Display*, the usual and natural way to step through all the elements of a multidimensional array is to use *for* loops nested one inside another. Each *for* loop steps through one of the array indexes.

*nested  
for loops*

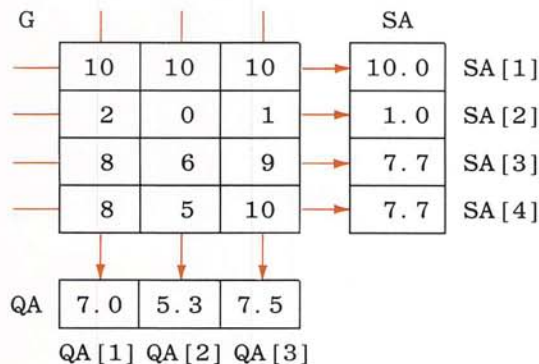
## Pitfall

### Exceeding Storage Capacity

It is very easy to use unreasonably large amounts of storage when programming with multidimensional arrays. Even a modest-looking multidimensional array declaration can sometimes cause the computer to use a huge amount of storage. For example, consider the following reasonable-looking array declaration:

```
var A: array[0 .. 50, 0 .. 50, 0 .. 50] of integer;
```

The compiler must allocate storage for  $51 \times 51 \times 51 = 132,651$  integers. Many computer installations simply will not have enough storage available to accommodate such a program.



**Figure 10.15**  
Relationship of  
arrays for grading  
program.



## TURBO Pascal

### Typed Constants (Optional)

*Typed constants* are a TURBO Pascal feature that does not exist in standard Pascal. Technically speaking, these typed constants are not really constants at all. They are initialized variables, that is, variables that have an initial value specified in the program heading. However, they can serve many of the same functions that constants do.

#### Program

```

program ShowGrades(input, output);
  {Reads quiz scores for each student into a two-dimensional array G.
  Computes the average score for each student and the average score for each quiz.
  Displays the quiz scores and the averages.}
  const NumStudents = 4;
        NumQuizzes = 3;
  type Score = 0 .. 10;
        StudentIndex = 1 .. NumStudents;
        QuizIndex = 1 .. NumQuizzes;
        GradeArray =
          array[StudentIndex, QuizIndex] of Score;
        StuAveArray = array[StudentIndex] of real;
        QuizAveArray = array[QuizIndex] of real;
  var G: GradeArray;
      SA: StuAveArray;
      QA: QuizAveArray;

  procedure ReadQuizzes(var G: GradeArray);
    {Postcondition: G[SNum, QNum] contains the score that
    student number SNum received on quiz number QNum.}
    var SNum: StudentIndex;
        QNum: QuizIndex;
  begin {ReadQuizzes}
    for SNum := 1 to NumStudents do
      begin {Student number SNum}
        writeln('Enter the ', NumQuizzes:3, ' quiz scores');
        writeln('for student number ', SNum:3);
        for QNum := 1 to NumQuizzes do
          read(G[SNum, QNum]);
        readln
      end {Student number SNum}
    end; {ReadQuizzes}

```

**Figure 10.16**  
Program using  
two-dimensional  
arrays.

```

procedure QuizAves(var G: GradeArray; var QA: QuizAveArray);
{Precondition: G[SNum, QNum] contains the score that
student number SNum received on quiz number QNum.
Postcondition: QA[QNum] contains the average score for quiz QNum.}
var SNum: StudentIndex;
    QNum: QuizIndex;
    Sum: integer;
begin{QuizAves}
  for QNum := 1 to NumQuizzes do
    begin{Quiz number QNum}
      Sum := 0;
      for SNum := 1 to NumStudents do
        Sum := Sum + G[SNum, QNum];
      {Sum contains the sum of the student scores for quiz QNum.}
      QA[QNum] := Sum / NumStudents
    end {Quiz number QNum}
  end; {QuizAves}

procedure StudentAves(G: GradeArray; var SA: StuAveArray);
{Precondition: G[SNum, QNum] contains the score that
student number SNum received on quiz number QNum.
Postcondition: SA[SNum] contains the average score for student SNum.}
var SNum: StudentIndex;
    QNum: QuizIndex;
    Sum: integer;
begin{StudentAves}
  for SNum := 1 to NumStudents do
    begin{Student number SNum}
      Sum := 0;
      for QNum := 1 to NumQuizzes do
        Sum := Sum + G[SNum, QNum];
      {Sum contains the sum of the quiz scores for student SNum.}
      SA[SNum] := Sum / NumQuizzes
    end {Student number SNum}
  end; {StudentAves}

procedure Display(var G: GradeArray; var SA: StuAveArray;
                  var QA: QuizAveArray);
{Precondition: G[SNum, QNum] contains the score that
student number SNum received on quiz number QNum.
QA[QNum] contains the average score for quiz QNum.
SA[SNum] contains the average score for student SNum.
Postcondition: The scores and averages are displayed on the screen.}
const Space = ' ';
var SNum: StudentIndex;
    QNum: QuizIndex;

```

**Figure 10.16**  
(continued)

```

begin{Display}
  {First display the student scores and student averages.}
  writeln('Student':10, 'Ave':5, Space:10, 'Quizzes');
  for SNum := 1 to NumStudents do
    begin{outer for loop}
      write(SNum:10, SA[SNum]:5:1, Space:4);
      for QNum := 1 to NumQuizzes do
        write(G[SNum, QNum]:5);
      writeln
    end; {outer for loop}

  {Next display the averages of the quizzes.}
  write('Quiz Averages =':15, Space:4);
  for QNum := 1 to NumQuizzes do
    write(QA[QNum]: 5:1);
  writeln
end; {Display}

begin{Program}
  ReadQuizzes(G);
  QuizAves(G, QA);
  StudentAves(G, SA);
  Display(G, SA, QA);
  writeln('Now you know the scores!')
end. {Program}

```

### Sample Dialogue

```

Enter the 3 quiz scores
for student number 1
10 10 10
Enter the 3 quiz scores
for student number 2
2 0 1
Enter the 3 quiz scores
for student number 3
8 6 9
Enter the 3 quiz scores
for student number 4
8 5 10

```

Student	Ave	Quizzes		
1	10.0	10	10	10
2	1.0	2	0	1
3	7.7	8	6	9
4	7.7	8	5	10
Quiz Averages =	7.0	5.3	7.5	

Now you know the scores!

**Figure 10.16**  
(continued)



Typed constants are declared like ordinary constants, except that you must specify the type of the constant in the manner illustrated by the second of the following sample constant declarations:

```
const Max = 5;
      Count: integer = 3;
      Start = 1;
```

In this example `Max` and `Start` are untyped (ordinary) constants of type `integer`. They cannot be changed and can only be used in places where a literal constant, such as 5, can be used. On the other hand, the typed constant `Count` is a variable of type `integer` and can be used anywhere that such variables are allowed. In particular, it can be changed, and it can be used as an actual *variable parameter* of type `integer`. Moreover, the variable `Count` has an initial value. If the *first* line of the program with this declaration is

```
writeln(Count);
```

then this will cause the value 3 to be written to the screen.

Since typed constants are really variables, they may not be used in type declarations. Type declarations such as the following are *not* allowed:

```
const Last: integer = 5;
type RealList = array[1..Last] of real; {NOT ALLOWED}
```

## TURBO Pascal

### Array Constants (Optional)

In standard Pascal all defined constants must be of a simple type, and so array constants are not allowed. TURBO Pascal allows more flexibility in naming constants. TURBO Pascal has been extended to allow typed constants, and typed constants can be of almost any type. In particular, TURBO Pascal allows typed constants which are arrays, as in the following example:

```
const A: array[1..5] of integer = (10, 20, 30, 40, 50);
```

This declares `A` to be an array and gives it an initial value. `A` will be an ordinary array variable and can have its value changed like any other array variable. However, `A` starts out with an initial value. If the program body opens as follows

```
begin{Program}
  for I := 1 to 5 do
    write(A[I], ' ');
```

then the output produced by this code will be

```
10 20 30 40 50
```

An important feature of array constants is that they may be used as actual variable parameters in procedure calls. This requires that the type be given a type name and that the constant be declared using the type name, as in the following example:

```
type List = array[0..3] of real;
const R: List = (0.0, 1.1, 2.2, 3.3);
```

With these declarations R may be an actual parameter of type List.

Unlike standard Pascal, TURBO Pascal does allow type declarations to precede constant declarations. In fact, TURBO Pascal allows the various kinds of declarations to appear in any order and even allows repeated sections of the same kinds of declarations, provided that all items are declared before they are used in any other declaration.

Multidimensional array constants are declared by enclosing the constants of each dimension in a separate set of parentheses, as illustrated by

```
type Matrix = array[1..2, 1..2] of integer;
const M: Matrix = ((1, 2), (3, 4));
```

This constant declaration sets the value of M[1, 1] equal to 1, that of M[1, 2] equal to 2, and those of M[2, 1] and M[2, 2] equal to 3 and 4, respectively.

---

The more abstract the truth is  
that you would teach, the more  
you have to seduce the senses  
to it.

*Nietzsche*

---

## Summary of Problem Solving and Programming Techniques

- When you are designing an algorithm, working with some specific sample data often helps clarify your thinking.
- Drawing a picture can often reveal the structure of a problem and lead to an algorithmic solution.
- You can often adapt a previously designed algorithm to solve a new, but similar, problem.
- One of the first tasks in designing an algorithm is to decide on a data structure to represent the data that will be manipulated by the algorithm.
- Data abstraction allows us to forget the inessential details of a data structure and concentrate on the substantive issues of algorithm design. In this way it is like, and in fact goes hand in hand with, procedural abstraction.
- If you need more than one value for each array index, you can use two or more parallel arrays.
- If you need a list of arrays, you can use an array of arrays.

- If you desire more than one index for each array element, you can use a multidimensional array type.
- Arrays are often processed sequentially. If the array is large, this can be time consuming. You can save time by terminating array processing as soon as the relevant information has been obtained. For example, when looking for something in an array, the processing can stop when the item or condition is found.
- One way to help find the correct index expression for array processing is to try specific numbers in place of the variables and see if the expression yields the correct value.

---

## Summary of Pascal Constructs

### array type declaration

Syntax:

```
type <name> =  
    array[<type 1>, <type 2>, . . . , <type n>] of <component type>;
```

Example:

```
type ArrayName =  
    array[0 .. 5, 'A' .. 'F'] of real;
```

The  $n$  index types must be ordinal types or subrange types. They may be either type names or type definitions. There may be any number of index types as long as there is at least one. The component type may be any Pascal type, including the defined types to be introduced in later chapters.

### packed arrays of characters (optional)

Syntax:

```
type <name> =  
    packed array[<type>] of char;
```

Example:

```
type CharString =  
    packed array[1 .. 20] of char;  
var A: CharString;
```

A packed array of characters is like an ordinary array of characters but with some additional properties: The compiler is instructed to use storage more efficiently; the entire array may be an argument to a `write` or `writeln` statement, as shown below. A string may be assigned to the array using a string constant and an assignment statement as shown below, provided the string has exactly the same number of characters as the array has indexes.

```
A := 'do be do to you'  
writeln(A)
```



## Exercises

### Self-Test Exercises

6. The variable *S* declared below can be used to hold a page or screen of text consisting of 20 lines with 50 characters per line. Give code to perform each of the tasks listed after the declarations.

```
type OneScreen = array[1 . . 20, 1 . . 50] of char;
var S: OneScreen;
```

- Write the first character of each line to the screen.
- Write the last line to the screen.
- Write the entire page of text to the screen.

### Interactive Exercises

7. Write a program to read in a list of 10 numbers and letters that are typed in one letter and one number per line. The program should then echo the input back in the same format. Use two parallel arrays to hold the data.

8. Write a program to fill a two-dimensional array *A* of the type shown below, display it to the screen in the natural way, and then allow the user to type in any pair of indexes *I*, *J* and have the program write out the value of *A* [*I*, *J*].

```
array[1 . . 3, 1 . . 2] of integer
```

Include a loop to let the user enter different values of *I* and *J* for as long as the user wishes. The procedures *ReadQuizzes* and *Display* from Figure 10.16 can be used as models for the general method of reading in the array and displaying the array. They will not work without changes, but they do give the general idea of what is to be done.

### Programming Exercises

9. Write a program that reads in a person's name in the format

<first name> <middle name or initial or nothing> <last name>

The program should then output the name in the format

<last name>, <first name> <middle initial>.

If the person inputs no middle name or initial, then the output should of course omit the middle initial. The program should work the same whether or not the user places a period after the middle initial. You may find it helpful to use more than one array. The program should work as long as each name is at most 10 characters long. If a name is over 10 characters long, then the program should use the first 10 letters of the name and should also issue an output statement saying that it did not use all of the letters in a

certain name. (There is no "<" or ">" in the input. They are just to make the instructions easier to read.)

10. If you have not yet done so, do Exercise 36 in Chapter 8. You can do this exercise either with or without the TURBO string functions and procedures. In particular, you can do the exercise even if you are not on a TURBO Pascal system. (If you are not on a TURBO Pascal system, you will probably want to read the optional section of Chapter 8 entitled "The Functions pred, succ, ord, and chr.")

11. Write a program that reads a line of text and then outputs a list of all the letters that occur in the text, together with the number of times each letter occurs in the line. The letters should be listed with the most frequently occurring letter given first, the next most frequently occurring letter given second, and so forth. Use two parallel arrays with the index type 1 . . . 26. One array will hold letters, and the other will record the number of times that the corresponding letter occurs. Your program should consider upper- and lowercase versions of a letter to be equal, and so the input line

DO be do

should produce output similar to the following:

letter:	number of occurrences:
d	2
o	2
b	1
e	1

12. Modify the history table program in Figure 10.6 so that it records the year of death as well as the year of birth for each composer. The output should show both years for each composer, and the list should be sorted by year of death. Composers who died in the same year should be ordered by year of birth. Composers who are still alive should be placed at the end of the list, ordered by year of birth.

13. Write a checkbook balancing program. The program will read in the following for all checks that were not cashed as of the last time you balanced your checkbook: the number of each check, the amount of the check, and whether or not it has been cashed yet. (Use three parallel arrays, one for the numbers, one for the amounts, and one to record whether the check was cashed.) The program also reads in the deposits, as well as the old and the new account balance. The new account balance should be the old balance, plus all deposits, minus all checks that have been cashed. The program outputs what the total of the checks cashed is, the total of the deposits, what the new balance should be and how much this figure differs from what the bank says the new balance is. It also outputs two lists of checks: the checks cashed since the last time you balanced your checkbook and the checks still not cashed. Both lists are sorted by check number.

14. Modify the automated drill program in Figure 10.7 so that it can account for families in which there is more than one composer. The most notable example is the Bach family. The very well known Johann Sebastian Bach had four sons who were composers: Wilhelm Friedemann Bach, Karl Philipp Emanuel Bach, Johann Christoph Freidrich Bach, and Johann Christian Bach. Your modified program looks for multiple matches,

and if it finds more than one match for, say "Bach," it tests to see whether the date matches any one of them. If a date does match, the program considers it a correct answer. If no dates match, it outputs all composers whose name contained the sub-pattern, along with the date for each composer.

15. A queue is a list that is used in a restricted way. Items are always added to the end of the list. For example, when X is added to A, B, C, the result is A, B, C, X. Items can only be removed from the front of the list. In order to remove C from the list, A and B must first be removed. Write a set of procedures for treating an array of characters as a queue. It should include procedures for insertion and deletion. The limits of the array index should be defined constants of type `integer`. Allow for the possibility that the number of elements in the queue is less than the size of the array.

16. Use parallel arrays in a program that reads in five playing cards and then displays the hand sorted for poker; that is, cards of the same value are grouped together (all twos are together, all threes are together, all kings are together), and the groups are then sorted by value (all twos first, then all threes, then all fours, etc.). Each card has a value (2 to 10, ace, king, etc.) and a suit (clubs, diamonds, spades, or hearts). Count aces as the highest card in the sorted list. A sample display is

```
2 of diamonds
2 of clubs
king of spades
king of hearts
ace of diamonds
```

17. Telephone dials and push-buttons have letters as well as numbers. Hence, some phone numbers spell words. For example 452-4357 is also 452-HELP. Write a program to help people find words for phone numbers. The program should read in the last four digits of a phone number and then output all possible letter versions of that number. For example, 4357 can be HELP. It can also be GDJP, as well as other letter combinations. Use three arrays indexed by 0 . . . 9 to hold the three letters that correspond to each number. There are no letters for 0 or 1. Do something graceful with those digits. If you prefer, you can use 1 . . . 9 or 2 . . . 9 as the array index type.

18. Write a program that does the following: asks the user to type in nine numbers in three rows of three numbers each, reads the numbers into a two-dimensional array, computes the sum of each row and each column, and then outputs the array as well as the row and column sums in the following format:

ARRAY:	ROW SUMS:
1 2 3	6
3 3 3	9
3 2 1	6
COLUMN SUMS:	
7 7 7	

19. Write a program to assign passenger seats in an airplane. Assume a small airplane with seats numbered as follows:



1	A	B	C	D
2	A	B	C	D
3	A	B	C	D
4	A	B	C	D
5	A	B	C	D
6	A	B	C	D
7	A	B	C	D

The program should display the seat pattern, marking with an 'X' the seats already assigned. For example, after seats 1A, 2B, and 4C are taken, the display should look like

1	X	B	C	D
2	A	X	C	D
3	A	B	C	D
4	A	B	X	D
5	A	B	C	D
6	A	B	C	D
7	A	B	C	D

After displaying the seats available, the program prompts for the seat desired, the user types in a seat, and then the display of available seats is updated. This continues until all seats are filled or until the user signals that the program should end. If the user types in a seat that is already assigned, the program should say that the seat is occupied and ask for another choice.

20. Write a program that accepts input like that accepted by the program in Figure 9.14 and then outputs a bar graph like the one in that figure with the exception that your program outputs the bars vertically rather than horizontally. A two-dimensional array may be useful.

21. A graph with NumVert vertices can be represented by an array of the type

```
var G: array[1 .. NumVert, 1 .. NumVert] of boolean;
```

If  $G[I, J]$  has value true, it means that there is an arc from node I to node J. Write a program that takes as input the array representation of a graph and two nodes in the graph, and that outputs a path from the first node to the second or else announces that no path exists. Assume that the arcs are "one-way" arrows, so that if there is an arc from I to J, then the path can go from I to J but not necessarily in the other direction.

22. The game of Life, invented by J. H. Conway, is played by choosing an arrangement of marks on a rectangular grid and watching them change according to the following rules: If two or three of the four immediately adjacent positions are marked, then the mark is left; otherwise, it disappears. An unmarked cell becomes marked if exactly three of its immediately adjacent positions are marked. Write a program that accepts a pattern and then shows the series of patterns it produces. Have the program stop after the pattern stabilizes or after 10 iterations if it does not stabilize by then. Use a grid size of at least 10 by 10. All changes occur simultaneously, and so the program will need two copies of the configuration, one for the old pattern and one for the new one.

23. Write a program that reads in a screen display consisting of 20 lines of 20 characters each and then rotates it in stages, displaying each stage as follows: First it is rotated onto its side, then it is rotated more until it is upside down, then it is rotated to three-quarters of the way around, and finally it is rotated back to its original orientation.
24. An  $n$  by  $m$  matrix is a rectangular array of numbers. For example, the following is a 4 by 3 matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 5 & 9 \\ 1 & 3 & 9 \\ 6 & -3 & 5 \end{pmatrix}$$

The entries of a matrix are normally numbered by two subscripts, one for the row and one for the column. The following illustrates the numbering of a 4 by 3 and a 3 by 2 matrix:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{pmatrix} \quad \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$$

The product of an  $m$  by  $n$  matrix with entries  $a_{ij}$  and an  $n$  by  $p$  matrix with entries  $b_{ij}$  is an  $m$  by  $p$  matrix whose entries  $c_{ij}$  are defined as follows:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

Write a program that reads in an  $m$  by  $n$  matrix row by row, then reads in an  $n$  by  $p$  matrix row by row, and then computes the product matrix and displays the two matrices as well as their product matrix on the screen. Use integer values for the matrix entries. Use 3, 4, and 5 for the values of  $m$ ,  $n$  and  $p$ , but declare constant names for them so that they can easily be changed.

25. Write a program that reads an  $n$  by  $n$  matrix into a two-dimensional array A and then determines which, if any, of the following special classes the matrix falls into:

Symmetric:  $A[I, J] = A[J, I]$  for all indexes I and J.

Diagonal:  $A[I, J] = 0$  whenever I and J are different.

Upper triangular:  $A[I, J] = 0$  whenever  $I < J$ .

Lower triangular:  $A[I, J] = 0$  whenever  $I > J$ .

Use 6 as the value of  $n$ .

26. Write a program that allows two users to play tic-tac-toe. The program should ask for moves alternately from player X and player O. The program displays the game positions as follows:

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

The players enter their moves by entering the position number. After each move, the program displays the changed board. A sample board configuration is

X	X	O
4	5	6
O	8	9

27. Redo the previous exercise, but this time have the computer be one of the two players.
28. Write a program that allows the user to make a pattern on the screen using the keyboard and then stores the pattern in a two-dimensional array and echoes it back to the user. It continues to do this until the user indicates that the program should end. Use any array dimensions that are convenient, but allow at least a four by four pattern of characters.
29. If you have not yet done so, do Exercise 34 in Chapter 8. You can do this exercise either with or without the TURBO string functions and procedures. In particular, you can do the exercise even if you are not on a TURBO Pascal system.
30. If you have not yet done so, do Exercise 35 in Chapter 8. You can do this exercise either with or without the TURBO string functions and procedures. In particular, you can do the exercise even if you are not on a TURBO Pascal system.

---

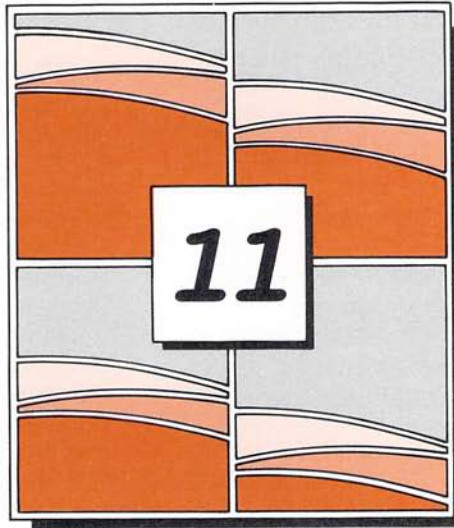
## References for Further Reading

D.F. Stubbs and N.W. Webre, *Data Structures with Abstract Data Types and Pascal*, 1985, Brooks/Cole. See Chapter 1 and the sections of Chapter 2 that cover arrays.

---







## ***Records and Other Data Structures***

'The time has come' the Walrus said,  
'To talk of many things:  
Of shoes—and ships—and sealing wax—  
Of cabbages—and kings. . . .'  
*Lewis Carroll, Through the Looking-Glass*

## Chapter Contents

Introduction to Records	Case Study—Sorting Records
Comparison of Arrays and Records	Searching by Hashing (Optional)
The Syntax of Simple Records	Variant Records (Optional)
Self-Test Exercises	Simple Uses of Sets
Records within Records	TURBO Pascal Pitfall—Limitations on Sets
Arrays of Records	More about Sets (Optional)
Sample Program Using Records	TURBO Pascal—Defined Set Constants (Optional)
Choosing a Data Structure	Summary of Problem Solving and Programming Techniques
The With Statement	Summary of Pascal Constructs
Pitfall—Problems with the With Statement	Exercises
TURBO Pascal—Use of Strings in Records	References for Further Reading
TURBO Pascal Case Study—Sales Report	
Case Study—Strings Implemented as Records	

In Chapters 9 and 10, we discussed the notion of a structured data type and introduced arrays as examples of structured types. In this chapter we introduce two other classes of structured data types, one called *record types* and another called *set types*. Record data types allow us to combine data of different types to obtain a single complex type that packages diverse subelements into a single (compound) value. Set data types let us combine items of the same type, but, unlike arrays, they do not order the items in any way. We also discuss techniques for using these data types as well as ways of deciding which of all the data types we have seen is best suited to a particular application.



---

## Introduction to Records

Sometimes it is useful to have a single name for a collection of values that may be of diverse types. For example, an inventory for a mail-order house might contain the following entry:

```
atomic can opener  
item #2001  
price $1,999.99
```

In this example, each inventory record consists of a name, a stock number, and a price. Although the record is conceptually a unit, the components are items of different data types. In Pascal, it is possible to define a structured data type consisting of a number of components, each of a possibly different type. These kinds of structured types are called *records*.

The individual components of a record are commonly referred to as *fields* or *components* or *component fields*. Each field has a name called a *field identifier*, which is some identifier chosen by the programmer when the record type is declared. Each field also has a type, which is specified when the record type is declared. A possible record declaration for the inventory record mentioned in the previous paragraph is the type `StockItem` defined below:

*component  
field*

```
type StockItem =  
    record  
        Name: array[1..20] of char;  
        Number: integer;  
        Price: real  
    end;
```

Name, Number, and Price are the field identifiers. `StockItem` is the name of the type. In a program with this type declaration, variables of type `StockItem` are declared in the usual way:

```
var Item1, Item2: StockItem;
```

A value of a record type is a collection of values that are each of some simpler type. The record has one value in the collection for each field identifier in the type declaration. The type of each of these values is the type specified for that field in the record type declaration. For example, a value of type `StockItem` is composed of three simpler values: One is an array of characters, one is of type `integer`, and one is of type `real`. These individual values are called the *component values* of the record. A sample record of type `StockItem` is illustrated in Figure 11.1.

*component  
value*

The field identifiers of a record are similar to the indexes of an array. They provide a way to name each individual value in the collection of values that make up the record. By adding the field name to a record variable, we can specialize the variable to one of its components. A component of a record variable is, as you might expect, called a *component variable*. To specify a component variable, we append a period and the field identifier to the record variable. (The period is usually pronounced “dot.”) For

*component  
variable*

---



instance, the component variable of `Item1` named by the field identifier `Price` is written as follows:

```
Item1.Price
```

It is a variable of type `real` and can be used just like any other variable of type `real`. For example, the following will write the real value 9.95 on the screen:

```
Item1.Price := 9.95;  
writeln(Item1.Price);
```

Similarly, the component variable `Item1.Number` is a variable of type `integer`, and the component variable `Item1.Name` is an array of characters. These component variables are illustrated in Figure 11.2.

When reading expressions involving records and arrays, always proceed from left to right. For instance, consider the expression

```
Item1.Name[4]
```

The identifier `Item1` names a record variable of type `StockItem`. By adding the field identifier, we specify a particular component variable. So `Item1.Name` denotes the component variable called `Name`. As specified in the type declaration, that is an array, and so we can add an array index to it. The complete expression thus refers to the fourth indexed variable of this array. To be very concrete, it refers to the fourth letter in the name of the item in this inventory record. This notation is pictured in the last illustration of Figure 11.2.

Figure 11.3 presents a Pascal procedure that fills a record variable of type `StockItem` and illustrates the basic technique for manipulating record variables.

A value of a record type is composed of a group of values of the component types specified in the declaration of that record type. This group can sometimes be treated as a single (structured) value. For example, a procedure parameter of a record type is written without any field identifiers. The situation is similar to that of arrays. Also, the value of an entire record variable can be set by a single assignment statement.

For example, suppose that `Item1` and `Item2` are record variables of the type `StockItem`, declared in Figure 11.3. The following two statements consist of a procedure call followed by an assignment:

```
ReadRecord(Item1);  
Item2 := Item1
```

If the declarations are as we gave them in Figure 11.3, then the first statement is a procedure call that sets the value of the record variable `Item1` by having the user enter the value of each component variable. The second statement sets the value of each component variable of `Item2` equal to the value of the corresponding component of `Item1`.

A data type is specified by describing both the values that can be held by the variables of that type and the operations that are allowed on those values. In the case of records, the values are collections of simpler values, each indexed by a field identifier. The "dot" operation combines a value and a field identifier to obtain a component value. For example, the value of the record variable `Item1` and the field identifier

*records as  
single values*

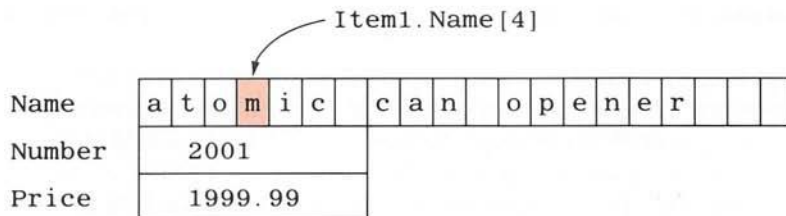
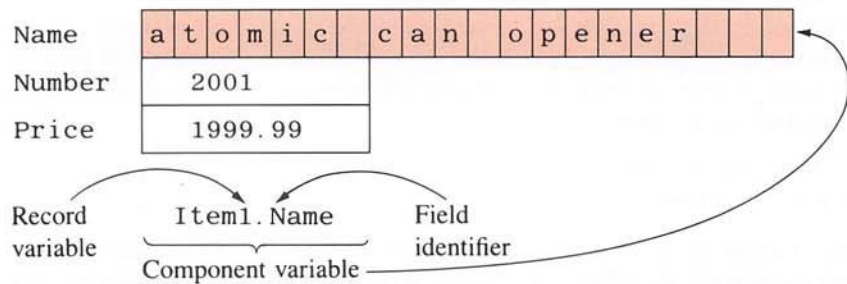
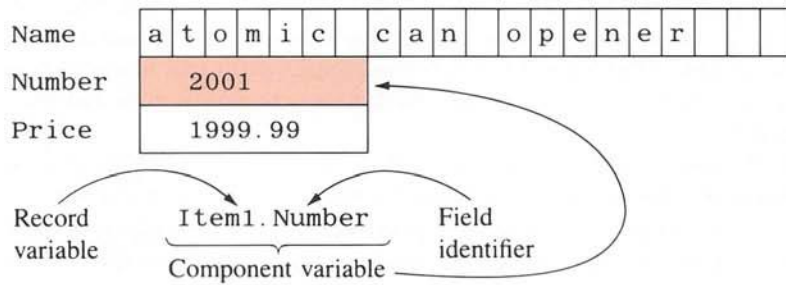
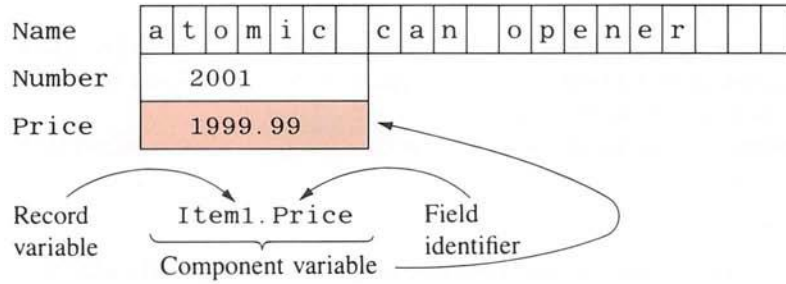
*operations  
on records*



```

type StockItem =
    record
        Name: array[1..20] of char;
        Number: integer;
        Price: real
    end;
var Item1: StockItem;

```



**Figure 11.2**  
Component  
variables.

```
type StockItem =
    record
        Name: array[1..20] of char;
        Number: integer;
        Price: real
    end;

procedure ReadRecord(var Item: StockItem);
{Sets the value of each component variable of Item
to a value read from the keyboard.}
var I: integer;
begin{ReadRecord}
    writeln('Enter name of item. Add blanks');
    writeln('to make it 20 characters long. ');
    writeln('Extra blanks are OK. ');
    for I := 1 to 20 do
        read(Item.Name[I]);
    readln;

    writeln('Enter stock number. ');
    readln(Item.Number);

    writeln('Enter price. ');
    writeln('Do not include a dollar sign. ');
    readln(Item.Price)
end; {ReadRecord}
```

**Figure 11.3**  
**Procedure to fill a**  
**record variable.**

Price can be combined to obtain the component variable `Item1.Price`, which contains the component value specified by the name `Price`. Additionally, the usual operations allowed on variables may be used with record variables, such as `Item1`, and with component variables, such as `Item1.Price`. Either may be used with the assignment statement, and either may be used as a parameter to a procedure. All this makes a record sound very much like an array. As we will see in the next section, arrays and records are very similar, but they also have important differences.

---

## Comparison of Arrays and Records

Records and arrays are similar in many ways. They both provide a way to give a single name to a collection of values. They both refer to elements of the collection by means of some sort of name. In the case of an array, the name is an index. In the case of a record, the name is a field identifier. A variable of either type can be thought of as a collection of variables of the component types.

On the other hand, arrays and records do have some important differences. The

---

elements in an array list must all be of the same type. The component values of a record may be of different types. The index of an array may be computed by the program. If the index type is  $1 \dots 50$ , then a variable of this type may be used as the index. So the name of an array index variable (such as  $A[I]$ ) may be computed by the program (by computing the value of  $I$ ). The name of a component variable of a record (such as  $Item1.Price$ ) must include a field identifier (such as  $Price$ ), and there is no way for the program to compute a field identifier. The programmer must write it into the program.

---

## The Syntax of Simple Records

Type definitions of records follow the pattern of the inventory record in the opening section of this chapter. The list of field identifiers is enclosed within the reserved words *record* and *end*. Each field identifier is followed by a colon and the type of that field. A component type may be any Pascal type. In particular, it can be a structured type such as an array or even another record type. The various field identifier parts are separated by semicolons. If two successive field identifiers are of the same type, their declarations may be combined by separating them with commas and only listing the component type once. For instance, the following two type declarations are equivalent:

```

type Employee =
    record
        Number: integer;
        BaseRate: real;
        OvertimeRate: real
    end;

type Employee =
    record
        Number: integer;
        BaseRate, OvertimeRate: real
    end;

```

Of course, you may not use the same field identifier to name two different fields within the same record type. However, field names from two different record types may be the same. Thus, the following is allowed:

```

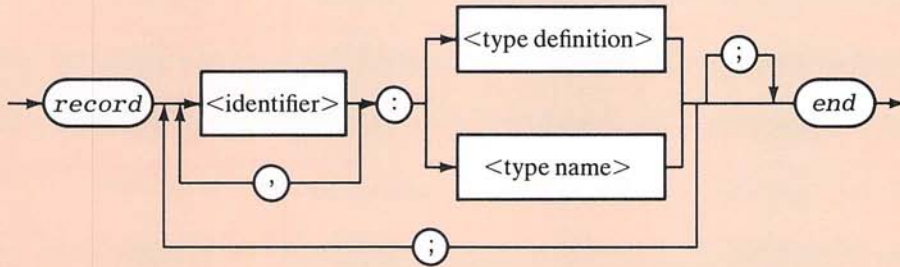
type Item =
    record
        Number: integer;
        Price: real
    end;

Temperature =
    record
        Number: real;
        Scale: char
    end;

```

---





**Figure 11.4**  
Syntax for simple  
record type  
definitions.

The syntax for type definitions is summarized in Figure 11.4. (As indicated there, you may, as usual, insert an extra semicolon after the last component, or you can omit it, as we have been doing in our examples.)

## Self-Test Exercises

1. Consider the following type declarations:

```

type Sample = record
    F1: integer;
    F2: char
end;
var X1, X2: Sample;

```

What will the output of the following piece of code be (provided that it is embedded in a complete Pascal program that includes the preceding type declarations)?

```

X1.F1 := 5; X1.F2 := 'A';
writeln(X1.F1, X1.F2);
X2 := X1;
X2.F1 := 6;
writeln(X2.F1, X2.F2)

```

2. Write a type declaration for a record type called Sam with one field of type integer, one of type real, and one of type char.
3. Write a program to fill (with data read from the keyboard) one record of the type described in the previous exercise and then display the record to the screen.

4. Write a type declaration for a student record that contains one field for the name; room for 10 quiz scores between 0 and 10; a midterm, a final exam score, and a final numeric grade, all in the range 0 to 100; and also a final letter grade.
- 

## Records within Records

Sometimes it makes sense to structure a record in a hierarchical way by making some component or components themselves records. For example, a record to hold the name and birth date of an individual might be of the following type:

```
type Date = record
    Month: 1 . . 12;
    Day: 1 . . 31;
    Year: integer
end;
Info = record
    Name: array[1 . . 20] of char;
    Birthday: Date
end;
```

A record variable might then be declared as follows:

```
var Person: Info;
```

If this record variable `Person` has its value set to record a person's birth date, then the year he or she was born is

```
Person.Birthday.Year
```

As always, the way to read these expressions is very carefully and from left to right. `Person` is a record variable. The component with field name `Birthday` is

```
Person.Birthday
```

This component variable is itself a record of type `Date`. Hence, it has three components, one of which is called `Year`.

Unfortunately, such complicated expressions are confusing. Fortunately, they can frequently be avoided by the use of procedure calls that manipulate entire records.

---

## Arrays of Records

Simple record types are seldom used by themselves. More often, records are grouped into larger units, such as arrays of records. For example, if `StockItem` is the record type defined in our introduction to records, then we are likely to use the records in an array of records. If there are 100 items in the inventory, a likely additional declaration would be the following:

---

```
type List = array[1..100] of StockItem;  
var Inventory: List;
```

The complete list of the inventory could then be read into the array by the following code:

```
for I := 1 to 100 do  
  begin  
    writeln('Next item: ');  
    ReadRecord(Inventory[I])  
  end
```

The procedure ReadRecord is given in Figure 11.3.

When you are programming with these nested structures, such as arrays of records, expressions can sometimes get quite complicated. To interpret them correctly, you must patiently work your way through them from left to right. As an example, review the declaration of the type StockItem and then try to figure out the following expression before reading on:

*syntax for  
nested  
structures*

```
Inventory[2].Name[3]
```

The expression is interpreted as follows: Inventory is an array of records of type StockItem. Hence, each indexed variable of that array is a record of that type. Inventory[2] is the second indexed variable of this array and hence is a record variable of type StockItem. Since it is of this record type, it makes sense to refer to the component called Name. The way to specify a component variable of any record variable is to append a period followed by the field name. In this case, the following is the component variable of the record variable Inventory[2] that has the field name Name:

```
Inventory[2].Name
```

The component named Name is an array of characters. Hence, the above is an array of characters whose third indexed variable is

```
Inventory[2].Name[3]
```

This expression is an indexed variable of type char. Its value is therefore the third letter in the name of the second item on the inventory list.

In order to truly master records and data structures, you must be able to unravel and understand expressions such as the one we just discussed. However, once you have mastered the notation, it is often best to avoid such complicated expressions. By treating a record as a unit and by using procedures that manipulate these units, we can simplify both our reasoning and our notation. The loop code given earlier in this section filled an array of records but did not need to mention any details about record fields. It filled record number I of the array with the single simple procedure call

*records  
as a unit*

```
ReadRecord(Inventory[I])
```



## Sample Program Using Records

As a simple example of how arrays of records are used, Figure 11.5 presents a simplified version of the grading program in Figure 10.16. This time we use an array of records rather than parallel arrays. To simplify the example, this version does not compute class averages for the quizzes.

### Program

```

program QuizAve2(input, output);
{Reads quiz scores for each student into an array of records;
computes each student's average and stores it in the record; displays each
student number followed by the student average followed by a list of quiz scores.}
const NumStudents = 4;
      NumQuizzes = 3;
type Score = 0 .. 10;
      StudentIndex = 1 .. NumStudents;
      QuizIndex = 1 .. NumQuizzes;
      Student =
        record
          Quiz: array[QuizIndex] of Score;
          Ave: real
        end;
      GradeBook = array[StudentIndex] of Student;
var B: GradeBook;

procedure ReadQuizzes(var B: GradeBook);
{Postcondition: For each student number SNum, B[SNum].Quiz contains the quiz
scores for student SNum in indexed variables B[SNum].Quiz[1], B[SNum].Quiz[2], ... }
var SNum: StudentIndex;
    I: QuizIndex;
begin{ReadQuizzes}
  for SNum := 1 to NumStudents do
    begin{Student number SNum}
      writeln('Enter the ', NumQuizzes:3, ' quiz scores');
      writeln('for student number ', SNum:3);
      for I := 1 to NumQuizzes do
        read(B[SNum].Quiz[I]);
      readln
    end {Student number SNum}
  end; {ReadQuizzes}

```

**Figure 11.5**  
Grading program  
using records.

```

procedure CompAverage (var B: GradeBook);
{Precondition: B contains the quiz scores for student SNum in
indexed variables B[SNum].Quiz[1], B[SNum].Quiz[2], . . .
Postcondition: B contains the average quiz score for student SNum in B[SNum].Ave.}
var SNum: StudentIndex;
    Sum, I: integer;
begin{CompAverage}
  for SNum := 1 to NumStudents do
    begin{Student number SNum}
      Sum := 0;
      for I := 1 to NumQuizzes do
        Sum := Sum + B[SNum].Quiz[I];
      {Sum contains the sum of the quiz scores for student SNum.}
      B[SNum].Ave := Sum/NumQuizzes
    end {Student number SNum}
  end; {CompAverage}

procedure Display (var B: GradeBook);
{Precondition: Scores and average for student SNum are in B[SNum].
Postcondition: The scores and average are displayed on the screen.}
const Space = ' ';
var SNum: StudentIndex;
    I: QuizIndex;
begin{Display}
  writeln( 'Student':8, 'Ave':5, Space:4, 'Quizzes');
  for SNum := 1 to NumStudents do
    begin{outer for loop}
      write(SNum:8, B[SNum].Ave:5:1, Space:4);
      for I := 1 to NumQuizzes do
        write(B[SNum].Quiz[I]:3);
      writeln
    end {outer for loop}
  end; {Display}

begin{Program}
  ReadQuizzes (B);
  CompAverage (B);
  Display (B);
  writeln('Now you know the scores!')
end. {Program}

```

#### Sample Dialogue

```

Enter the 3 quiz scores
for student number 1
10 10 10

```

**Figure 11.5**  
(continued)

```

Enter the 3 quiz scores
for student number 2
1 0 0
Enter the 3 quiz scores
for student number 3
7 8 5
Enter the 3 quiz scores
for student number 4
9 7 9
Student      Ave      Quizzes
      1  10.0      10 10 10
      2   0.3       1  0  0
      3   6.7       7  8  5
      4   8.3       9  7  9
Now you know the scores!

```

**Figure 11.5**  
(continued)

## Choosing a Data Structure

*data  
structures*

A *data structure* is a way of organizing data values. The various structured types that we have seen, such as arrays and records, are all data structures. Even simple variables of types such as integer or char could be considered data structures, although of a particularly simple kind. We now have a number of data structures available. We will eventually learn how to create other data structures within a Pascal program. When you are designing a program, the choice of an appropriate data structure is a critically important part of the design process.

*parallel  
arrays versus  
arrays of records*

Not infrequently, we have a choice of structures. As an example, we presented two versions of a grading program. The one in Figure 10.16 uses parallel arrays as the data structure. The one in Figure 11.5 uses an array of records as the data structure. The array-of-records data structure is closer to our intuition of how grade information is naturally kept and organized, and so it is often preferable to the two parallel arrays G and SA, which we used for the same data in Figure 10.16. This is a general phenomenon. Whenever parallel arrays are appropriate, an array of records is often also appropriate and usually is a preferable data structure.

In some situations, other array structures are more appropriate than an array of records. The program in Figure 10.16 uses a two-dimensional array and computes two kinds of averages: student averages and quiz averages. The program in Figure 11.5 uses an array of records but computes only student averages. With the two-dimensional array, we have a symmetry between students and quizzes. With the array of records, it is very easy to do calculations for each student, but calculations for one particular quiz are awkward. Which data structure we choose depends on the problem. For the output in Figure 10.16, a two-dimensional array and two one-dimensional arrays served best. For the simpler task in Figure 11.5, an array of records is more natural.



When you are designing a program, the choice of a data structure can be just as important as the designing of an algorithm for the program. The efficiency and clarity of a program can depend heavily on what data structures are used. Unfortunately, there is no algorithm for choosing a data structure. There are, however, a few useful guidelines.

Always consider the possibility of alternative data structures. Just because you find one that works does not mean that you have found the best one. All other things being equal, choose the one that is easiest to understand and manipulate.

Hierarchical data structures, like hierarchical control structures, make a program easier to understand. It pays to combine the basic data-structuring techniques to obtain hierarchical structures such as arrays of records, records of arrays, arrays of records of arrays, and so forth.

*hierarchical  
data structures*

There are some rules that apply to choosing between the various options for array types and/or record types. If all the items to be stored are of the same simple type, then a single array with that simple type as its base type can be used. If the items are of different types, then an array-of-records type can be used. An alternative to the array-of-records data structure is to use parallel arrays, that is, a collection of arrays with the same indexes. Some other programming languages do not have record types. When one is programming in these languages, parallel arrays are an even more important data structure.

---

## The With Statement

Look back at the procedure `CompAverage` given in Figure 11.5. The entire procedure deals with the single record named `B[SNum]`. Every field identifier refers to `B[SNum]`. It would be convenient to have a way to say that all references to a record of type `Student` are references to the record `B[SNum]`. Then we could simply write the field identifiers and not have to write `B[SNum]` each time. The *with* statement lets us do just that. But before dealing with arrays of records, as in the procedure `CompAverage`, let us look at the *with* statement within a simpler context.

Consider the following declarations:

```
type Sample = record
    F1: integer;
    F2: char
end;
var X: Sample;
```

The following code uses a *with* statement:

```
with X do
begin
    F1 := 5; F2 := 'A'
end;
writeln(X.F1, X.F2)
```

---

In the statement following *with X do*, all references to component field names F1 and F2 refer to X.F1 and X.F2, respectively. Hence, if the preceding code is embedded in a complete program, it will produce the following output:

```
5 A
```

Having mastered a simple example, we can now use a *with* statement to rewrite the procedure `CompAverage` so that the record name `B[SNum]` needs to be written only once. The procedure in Figure 11.6 does this and is equivalent to the one in Figure 11.5.

## Pitfall

### Problems with the With Statement

The syntax for the *with* statement is summarized in Figure 11.7. As indicated there, it is possible to have more than one record variable on the record variable list of a *with* statement. However, you must be careful to avoid any ambiguities when doing so. In particular, you cannot have two record variables of the same record type on the record variable list. This is because there would be no way to tell which record variable a field identifier referred to.

Even if you use only one record name in a *with* statement, there can still be a serious style problem. When a field identifier is seen in a *with* statement, it may not be immediately clear that it is a record field and not a simple variable. Although a *with* statement can make program code shorter and less complex, it can sometimes make it more difficult to read. To ensure readability, the *with* statement should be used sparingly.

## TURBO Pascal

### Use of Strings in Records

Records frequently include a field that is a string. In all the record examples that we have seen thus far, we have used arrays of characters to represent strings. In TURBO Pascal it usually makes more sense to declare these fields to be of a *string* type.

A preferable TURBO Pascal type declaration for inventory records of the type we discussed at the start of this chapter would be

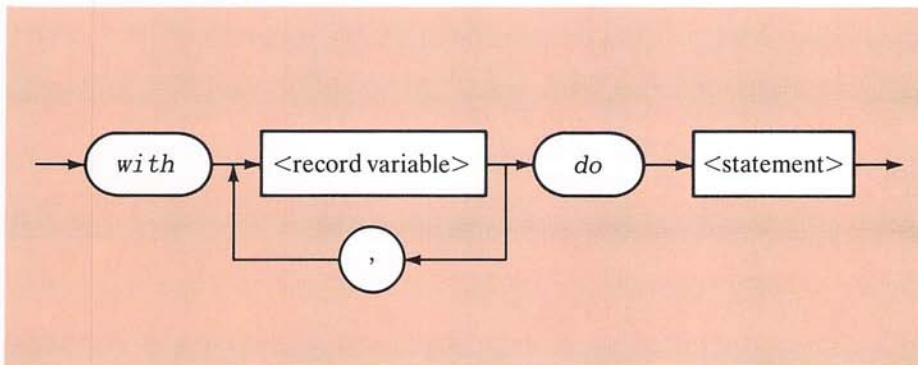
```
type NameString = string[20];
   StockItem =
       record
           Name : NameString;
           Number: integer;
           Price: real
       end;
```

```

procedure CompAverage (var B: GradeBook);
{Precondition: B contains the quiz scores for student SNum in
indexed variables B[SNum].Quiz[1], B[SNum].Quiz[2], . . .
Postcondition: B contains the average quiz score for student SNum in B[SNum].Ave.}
var SNum: StudentIndex;
    Sum, I: integer;
begin{CompAverage}
  for SNum := 1 to NumStudents do
    with B[SNum] do
      begin{Student number SNum}
        Sum := 0;
        for I := 1 to NumQuizzes do
          Sum := Sum + Quiz[I];
        {Sum contains the sum of the quiz scores for student SNum.}
        Ave := Sum/NumQuizzes
      end {Student number SNum}
    end; {CompAverage}
end;

```

**Figure 11.6**  
**Procedure**  
**CompAverage**  
**rewritten using**  
**with.**



**Figure 11.7**  
**Syntax of a with**  
**statement.**

## TURBO Pascal Case Study

### Sales Report

#### Problem Definition

The program we wrote for the Apex Plastic Spoon Manufacturing Company (in Chapter 9) has helped them to organize and optimize production. However, their annual company report shows that profits are falling despite increases in plant efficiency. The company decides that the problem is with their sales force and commissions us to write another, totally different program that will evaluate the performance of their sales force. For each salesperson, the program reads in the salesperson's total sales for the month and also the total value of goods that were sold by that person but were returned



for credit during the month. The program outputs the net yield (sales minus returns) for each member of the sales force as well as the differences between each person's net yield and the average net yield.

### Discussion

*data  
structures*

The first step is to decide on a data structure for the task. The information for each salesperson is most naturally kept in a record:

```
type NameString = string[20];
   PersonInfo =
       record
           Name: NameString;
           Sales, Return, Net, Comparison: integer
       end;
```

The data consists of a list of records, and so it is most naturally represented as an array of records:

```
type Index = 1 .. NumPeople;
   InfoList = array[Index] of PersonInfo;
```

### ALGORITHM

The problem breaks down in a standard way into three main subtasks:

1. GetData: Input the data.
2. Compute the results.
3. WriteTable: Output the results.

The data flow diagram is given in Figure 11.8, and, as indicated there, the task of computing the results breaks down into three smaller tasks:

- 2a. Compute the net yield for each person.
- 2b. Compute the average net yield.
- 2c. Compute the difference between each person's net yield and the average net yield.

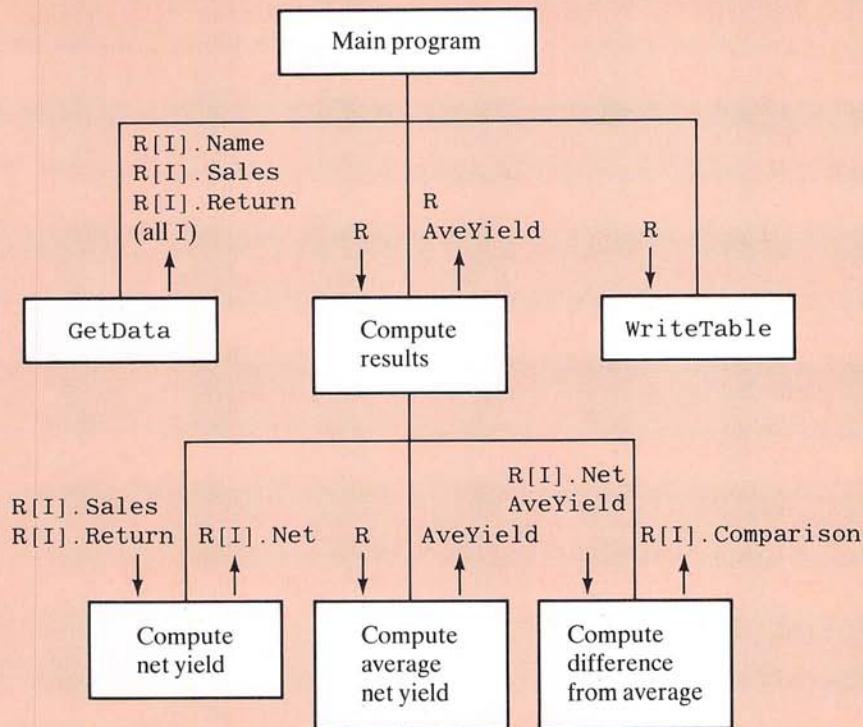
The complete program is given in Figure 11.9. Since subtasks 2a and 2b can each be performed by a loop that processes each record, and since they are each very simple, we have combined the two subtasks and used a single *for* loop to perform both calculations. Subtask 2c requires a separate loop.

## Case Study

### Strings Implemented as Records (Optional)

In the beginning of Chapter 10, we used the following type to represent strings as arrays of characters:

```
type CharString = array[1 .. MaxLength] of char;
```



### Data Summary

R: an array of records; each record has fields named  
 Name, Sales, Returns, Net, and Comparison.  
 AveYield: average net yield over all salespeople.

**Figure 11.8**  
**Data flow diagram**  
**for sales report**  
**generation.**

MaxLength is some declared constant. For concreteness we assumed that MaxLength is equal to 20. Typically the strings we encountered were less than the full 20 characters. Hence, when storing strings in such arrays, we must somehow account for the length of the string. In Chapter 10, we simply filled the unused array positions with blanks. This technique presents some problems when we process strings that end with a blank. If we fill the end of the array with blanks, then “Bach” and “Bach ” (the latter has one blank before the final quote) will be indistinguishable. In this section we will design a type that represents strings as records with two fields, one an array of characters and the other an integer giving the last array position used. This type will  
 (continued, page 446)

## Program

```

program SalesReport;
  {Reads in sales and return figures for each salesperson. Outputs a table showing
  for each salesperson: the input data plus net yield and comparisons to average net yield.
  Works in TURBO Pascal; needs some changes to work in standard Pascal.}
  const NumPeople = 3;
        Width = 12; {Field width}
  type NameString = string[20];
        PersonInfo =
          record
            Name: NameString;
            Sales, Returns, Net, Comparison: integer
          end;
        Index = 1 .. NumPeople;
        InfoList = array[Index] of PersonInfo;
  var R: InfoList;
      I: Index;
      Total, AveYield: integer; {AveYield is rounded to an integer.}

  procedure GetData(var R: InfoList);
    {Sets the fields Name, Sales, and Return for each record.
    Assumes that there are NumPeople records to enter.}
    var I: Index;
  begin{GetData}
    writeln('Enter names, sales, and returns, ');
    writeln('one person at a time: ');
    for I := 1 to NumPeople do
      with R[I] do
        begin{Fill R[I]}
          writeln('Enter name of salesperson: ');
          readln(Name);
          writeln('Enter Sales and Returns: ');
          readln(Sales, Returns)
        end {Fill R[I]}
      end;
    end; {GetData}

  procedure WriteTable(var R: InfoList; AveYield: integer);
    {Outputs a table showing each record in the array R.
    Also outputs the value AveYield labeled as the average net yield.}
    var I: Index;

```

**Figure 11.9**  
Sales report  
program.



```

begin{WriteTable}
  writeln('                      Sales Summary: ');
  writeln('Name');
  writeln('Sales' :Width, 'Returns' :Width,
          'Net' :Width, '+/-Average' :Width);
  for I := 1 to NumPeople do
    with R[I] do
      begin{Display R[I]}
        writeln(Name);
        writeln(Sales :Width, Returns :Width,
                Net :Width, Comparison :Width)
      end; {Display R[I]}
    writeln('Average net yield of all personnel =',
            AveYield:Width)
  end; {WriteTable}

begin{Program}
  GetData(R);
  Total := 0;

  {Compute net yields and average net yield (rounded to an integer).}
  for I := 1 to NumPeople do
    with R[I] do
      begin{Process R[I]}
        Net := Sales - Returns;
        Total := Total + Net
      end; {Process R[I]}
    AveYield := round(Total/NumPeople);

  {Compute amount above or below average.}
  for I := 1 to NumPeople do
    R[I].Comparison := R[I].Net - AveYield;

  WriteTable(R, AveYield)
end. {Program}

```

### Sample Dialogue

Enter names, sales, and returns,  
 one person at a time:  
 Enter name of salesperson:  
**Emanuel Transmission**  
 Enter Sales and Returns:  
**2000 100**  
 Enter name of salesperson:  
**Dusty Rhodes**

**Figure 11.9**  
 (continued)

Enter Sales and Returns:

**3000 200**

Enter name of salesperson:

**Chuck Steak**

Enter Sales and Returns:

**1000 500**

Sales Summary:

Name	Sales	Returns	Net	+/-Average
Emanuel Transmission	2000	100	1900	167
Dusty Rhodes	3000	200	2800	1067
Chuck Steak	1000	500	500	-1233
Average net yield of all personnel =				1733

**Figure 11.9**  
(continued)

(text continued from page 443)

accommodate trailing blanks as well as have other advantages over a simple array of characters.

Since TURBO Pascal has string types, this new type may seem superfluous to TURBO Pascal users. However, it is of some value to TURBO, as well as standard, Pascal users. This type will be usable in any version of Pascal. In standard Pascal it can be used as an equivalent of the TURBO Pascal type *string*[MaxLength]. In TURBO Pascal it can be used to write portable programs that will run on either TURBO Pascal or standard Pascal systems. It also gives some idea of how the TURBO Pascal string types might possibly be implemented.

*CharString*  
*redefined*

Our new definition of the type *CharString* is a record with two fields. One field is an array of characters that normally will be only partially filled. The other field is an integer variable that records the last array position used and so records the length of the string. The type declaration is as follows:

```

type CharString =
    record
        Symbols: array[1..MaxLength] of char;
        Length: integer
    end;

```

This is a new type, even though we are using the same type name, *CharString*. However, although it is a new type, it serves the same purpose as our old type *CharString* and so can be used to replace it. It is "the new improved *CharString* type!" Some sample string values of this redefined type are diagrammed in Figure 11.10.

To use this new type *CharString*, we must rewrite the basic string-manipulating procedures. These rewritten procedures are given in Figure 11.11. Notice that since

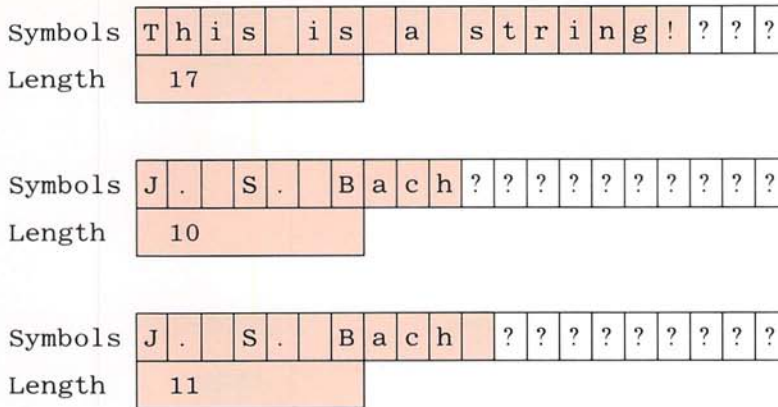


Figure 11.10  
Strings as records.

### Program

```

program TestProcedures (input, output);
{Tests the procedures StringReadln and StringWriteln
rewritten to work for the record version of the type CharString.}
const MaxLength = 20;
type CharString =
    record
        Symbols: array[1..MaxLength] of char;
        Length: integer
    end;
var Line: CharString;
    I: 1..MaxLength;
    Ans: char;

procedure StringReadln (var A: CharString);
{Fills the array A with one line of input characters. If there are
too many characters it discards all characters after the first MaxLength characters.}
var I: integer;
begin {StringReadln}
    I := 1;
    while (not eoln) and (I <= MaxLength) do
        begin {while}
            read (A.Symbols[I]);
            I := I + 1
        end; {while}
    if (not eoln) {and hence (I > MaxLength)} then
        writeln('Only ', MaxLength, ' characters read in. ');
    readln;

    A.Length := I - 1
end; {StringReadln}

```

Figure 11.11  
Strings  
reimplemented  
as records.  
(Optional)



```

procedure StringWriteln(var A: CharString);
{Writes the string A to the screen and advances to next line.}
var I: integer;
begin{StringWriteln}
    for I := 1 to A.Length do
        write(A.Symbols[I]);
    writeln
end; {StringWriteln}

begin{Program}
    repeat
        writeln('Enter a string and press return. ');
        for I := 1 to MaxLength do
            write(I mod 10:1); {To make it easy to count.}
        writeln;
        StringReadln(Line);
        writeln('The string array contains: ');
        StringWriteln(Line);
        writeln('More? (y/n) ');
        readln(Ans)
    until (Ans = 'n') or (Ans = 'N');
    writeln('End of test. ')
end. {Program}

```

### Sample Dialogue

```

Enter a string and press return.
12345678901234567890
Do Be Do
The string array contains:
Do Be Do
More? (y/n)
y
Enter a string and press return.
12345678901234567890
Life is a jelly doughnut. Isn't it?
Only      20 characters read in.
The string array contains:
Life is a jelly dough
More? (y/n)
n
End of test.

```

**Figure 11.11**  
(continued)

we now keep a record of the length of a string, we need not fill unused symbol positions with blanks. The new type `CharString`, as well as the new basic procedures `StringWriteln` and `StringReadln`, can be used anywhere that we used the old ones defined in Figure 10.1.

---

## Case Study

---

### Sorting Records

In Chapter 9 we developed a method for sorting an array of integers. In Chapter 10 we adapted that algorithm to a sorting algorithm for parallel arrays. In this section we will adapt the original sorting algorithm from Chapter 9 again. This time we will derive an algorithm to sort records according to the values in one particular field. For example, you may want to sort inventory records by price from the least expensive to the most expensive. We will adapt the sorting procedure from Figure 9.16 so that it sorts inventory records according to price.

#### Problem Definition

Before we begin to adapt our sorting procedure, let us be sure that we understand exactly what kind of records our procedure will need to sort. Rather than using the inventory type that opened this chapter, we will use the following new type definition, in order to have yet another sample type:

```
type NameString = string[20];
   Entry =
       record
           Name: NameString;
           Quantity: integer;
           Price: real
       end;
   EntryList = array[Index] of Entry;
```

The above type declaration works in TURBO Pascal. In standard Pascal you could use the type `CharString`, defined in the previous case study, in place of the type `NameString`.

Suppose that the array `B` is of type `EntryList`. We wish to sort the records in `B` so that the price fields of the records are in increasing order. If the first array index is 1, then we want the following to hold:

$$B[1].Price \leq B[2].Price \leq B[3].Price \dots$$

### Discussion

In Chapter 9 we used the following algorithm to sort a simple array of integers A.

```
for I := <first index> to Last - 1 do
    Exchange (A[I], A[<suitable index>])
```

where Last records the last array index used.

To adapt this code to an array of records B, we need to do the following:

1. Replace the simple array A with an array of records B.
2. Write a procedure like Exchange that applies to records of type Entry.
3. Design an algorithm to compute the <suitable index>.

Tasks 1 and 2 are trivial. The version of Exchange that applies to records is called RecordExchange. It is shown in Figure 11.13, and it is exactly the same as the procedure Exchange except that the type integer is replaced by the type Entry.

B[1]	B[2]	B[3]
Coffee	Tea	Milk
30	50	20
6.98	3.95	0.97

Find the smallest value of B[I].Price

B[1]	B[2]	B[3]
Milk	Tea	Coffee
20	50	30
0.97	3.95	6.98

Interchange entire records

**Figure 11.12**  
Test and  
interchange for  
sorting records.

### Imin

Task 3 will be accomplished by adapting the procedure Imin from Figure 9.16. The version shown there worked for an array of integers A, and so it compared array values A[I]. This version will apply to entire records B[I] in our array of records, but only the Price field will be used to determine the ordering. Hence, comparisons involving the record B[I] are made using B[I].Price. However, once the function Imin determines the <suitable index>, procedure RecordExchange interchanges the entire record B[I] with the entire record indexed by <suitable index>. This distinction between testing one field and moving the entire record is illustrated in Figure 11.12. Since it is irrelevant to the issues at hand, the internal structure of the strings is not diagrammed. The sorting procedure is given in Figure 11.13.



## Searching by Hashing (Optional)

Suppose we want to store inventory records so that we can later retrieve any particular record by specifying its stock number. To be specific suppose the records are of the following form, with the stock number stored in the Number field:

```
type Item =
  record
    Number: integer;
    Price: real;
    Rating: char
  end;
```

If the records have Number fields with the values 1 through 50, then we could store them in an array of the following type, placing record number  $N$  in indexed variable  $A[N]$ :

```
var A: array[1 .. 50] of Item;
```

The record numbered  $N$  can be retrieved immediately since we know it will be in  $A[N]$ .

But suppose the numbers do not form a neat range like 1 . . 50. Suppose that we know there will be 50 or fewer numbers but that they will be distributed in the range 1 through 5000. We could use an array with index type 1 . . 5000, but that would be extremely wasteful of storage since only a very small fraction of the indexed variables would be used. Hence, it appears that we have no alternative but to store the records in an array with 50 elements and to use a serial search through the array whenever we wish to find a particular record. Things are not that bad. If we are clever we can store the records in an array with relatively few (well under 200) indexed variables and yet retrieve records much faster than we would by serial search. In fact, there is more than one way to achieve this goal. In Chapter 14, we will present one method that depends on the array being sorted. In this section we will present a method that does not require sorting the array.

To illustrate the trick involved, suppose that somebody tells us that the 50 numbers will turn out to be the following:

100, 200, 300, 400, ..., 4900, 5000

In that case, we can store the records in an array  $A$  with index type 1 . . 50. The record with number field equal to  $N$  is stored in indexed variable

$A[N \text{ div } 100]$

With the aid of the *div* operation, we can make do with the index type 1 . . 50 even though the numbers become as large as 5000. If we want record number 700, we compute  $700 \text{ div } 100$  and obtain the index 7. The record is stored in indexed variable  $A[7]$ .

```

const High = 10;
type NameString = string[20]; {In standard Pascal, use one of the array
                                types we called CharString in place of this type.}
Index = 1 .. High;
ExtendedIndex = 0 .. High;
{To be consistent with earlier work, we are using 0 to indicate
the empty list. This is not important to the issues in this figure.}
Entry =
    record
        Name: NameString;
        Quantity: integer;
        Price: real
    end;
EntryList = array[Index] of Entry;

procedure RecordExchange(var X, Y: Entry);
{Interchanges the values of X and Y.}
var Temp: Entry;
begin{RecordExchange}
    Temp := X;
    X := Y;
    Y := Temp
end; {RecordExchange}

function Imin(var B: EntryList; Start: Index; Last: Index): Index;
{Returns the index I such that B[I].Price is the smallest of the values:
B[Start].Price, B[Start + 1].Price, . . . B[Last].Price.}
var I: integer;
    Min: real;
begin{Imin}
    Imin := Start; {tentatively}
    Min := B[Start].Price; {minimum so far}
    for I := Start + 1 to Last do
        if B[I].Price < Min {Test only the Price} then
            begin{then}
                Min := B[I].Price;
                Imin := I
                {Min is the smallest of the values B[Start].Price, . . . ,B[I].Price;
                the tentative value of Imin is x such that B[x].Price = Min.}
            end {then}
    end; {Imin}

```

**Figure 11.13**  
**Procedure for**  
**sorting an array of**  
**records.**

```

procedure RecordSort (var B: EntryList; Last: ExtendedIndex);
{Sorts the partially filled array B into increasing order of prices using
the selection sort algorithm. Postcondition: The array elements have been
rearranged so that B[1].Price <= B[2].Price <= . . . <= B[Last].Price.}
var I: Index;
begin {RecordSort}
  for I := 1 to Last - 1 do
    begin {Place correct value in B[I]}
      RecordExchange (B[I], B[Imin (B, I, Last)]) {Interchanges entire records}
      {B[1].Price <= B[2].Price <= . . . <= B[I].Price
       are the I smallest of the original array elements;
       the remaining elements are in the remaining positions.}
    end {Place correct value in B[I]}
  end; {RecordSort}

```

**Figure 11.13**  
(continued)

This general technique is called *hashing*. One field of the record type, called the *key*, is singled out to serve as the name of the record. In our example, this was the Number field. A function *F*, called the *hash function*, maps key values to array indexes. In our example, if *N* is the value of the Number field of a record, then that record is stored in array position  $A[F(N)]$ . The function *F* must be chosen so that  $F(N)$  is always within the index range of the array. The hash function *F* can be either a declared Pascal function or an arithmetic expression. In our example  $F(N)$  was  $N \text{ div } 100$ .

In our example, every *N* produced a different value of  $F(N)$ . That is perfect, but one can not always find a perfect hash function. Suppose that we change the example by substituting 399 for 400. Then record number 300 will be placed in  $A[3]$ , as before. Record number 399 is supposed to be placed in  $A[399 \text{ div } 100]$ ; in other words, record number 399 is also supposed to be placed in  $A[3]$ . There are now two different records that belong in  $A[3]$ . This situation is known as a *collision*. In this case we could redefine the hash function to avoid the collision. In practice, you usually do not know the exact numbers that will occur as record keys, so you can not design a hash function that is guaranteed to be free of collisions. Something must be done to cope with collisions.

Typically you do not know what numbers will be used as the key values, but you do know an upper bound on how many there will be. The usual approach is to use an array size that is larger than is needed, typically two to three times as large as the number of records to be stored. The extra array positions make collisions less likely. A good hash function will distribute the key values uniformly through the index range of the array. If the array index type is  $1 \dots 100$ , then you might use the following hash function to produce an array index for the record whose Number field is equal to *N*:

$$(N \bmod 100) + 1$$

*key*

*hash  
function*

*collisions*



(The likelihood of collisions can be reduced by using the pseudorandom number generator discussed in the optional section of Chapter 8 entitled “Designing Your Own Pseudorandom Number Generator.” For example, the following might be used as a hash function to obtain an index in the range 1 . . . 100:

$$(\text{Random}(N) \bmod 100) + 1$$

The function `Random` is defined in Figure 8.9.)

*copied  
with  
collisions*

One way to deal with a collision is with the following algorithm:

Given the key  $N$ ,

1. Compute the index  $F(N)$ .
2. If  $A[F(N)]$  does not already contain a record, then store the record in  $A[F(N)]$  and end the algorithm.
3. If  $A[F(N)]$  already contains a record, then use  $A[F(N) + 1]$ ; if that contains a record, use  $A[F(N) + 2]$ , etc. until a vacant position is found. (When the highest numbered array position is reached simply go to the start of the array. For example, if the index type is 1 . . . 100, and 99 is full, try 100, 1, 2, etc., in that order.)

This requires that the array be initialized so that the program can test to see if an array position already contains a record. For example, if the key will always be a positive integer, then the key field of each array element can be initialized to 0. As long as it has a value of 0, the program knows that it does not contain a record. The procedure `Initialize` in Figure 11.14 initializes an array in this way. Figure 11.14 also contains procedures to store and retrieve records using this technique. The next few paragraphs explain the details of how these procedures work.

*Insert*

The procedure `Insert` will insert record  $R$  into array  $A$  by hashing the key value  $R$ . `Number` and handling collisions as in the above algorithm. The heart of the procedure is the following. (the hash function is called `Hash`)

```

I := Hash(R.Number);
while (A[I].Number <> 0) {position occupied} do
  begin{Go to next array position.}
    I := (I mod MaxI) + 1
    {I := I + 1, but with wrap around at the end so MaxI + 1 is 1.}
  end; {Go to the next array position.}
{I is the first vacant position >= Hash(R.Number).}
A[I] := R

```

If everything went smoothly this could be used as the body of the procedure. However, you should always plan on things not going smoothly. Therefore, the procedure in Figure 11.14 includes a number of checks. It checks to see if the array already contains a record with the specified item number; if it does, then it tells this to the user. It also counts the loop iterations to see if it has inspected the entire array. If it inspects the entire array without finding a vacant position it announces that the array is full.

The procedure `Find` searches an array  $A$  to see if it contains a record whose `Number` field is equal to `Key`. It assumes that the array has been initialized with the proce-

procedure `Initialize` and then changed only by the procedure `Insert`. The heart of this procedure is

```

I := Hash (Key) ;
Count := 1; {Counts up to MaxI to detect if the entire array has been inspected.}
while (A[I].Number <> Key) {Key not found} and
      (Count < MaxI) {entire array not yet checked} do
  begin{Go to next array position.}
    I := (I mod MaxI) + 1;
    {I := I + 1, but with wrap around at the end so MaxI + 1 is 1.}
    Count := Count + 1
  end; {Go to next array position.}
{Either A[I].Number = Key or no record in the array has its Number field equal to Key.}

```

If the record is in the array then, at the end of the `while` loop, `I` will be set so that `A[I]` contains the record. If the record is not in the array, then the above code would search the entire array before it stops looking for the record. That is more work than is needed.

If the record is not in the array, then the loop can usually terminate much sooner. Recall that if there is a record with its `Number` field equal to `Key`, then it should be in position `Hash (Key)` or else in the first available position after that. The loop starts searching at array index `Hash (Key)`. Should it ever encounter a vacant position, it knows that: if the record were inserted, then it would have been inserted there or before there. Hence, it knows it has looked every place that the record could possibly be. The final version of the procedure `Search` tests for a vacant position and terminates the loop if it finds one.

Another technique for coping with collisions is to use two different hash functions. If a collision occurs, the program uses the second hash function to compute a number  $n$ , and then, rather than going to the next location, it goes to the location  $n$  places after the location given by the first hash function. This is called *double hashing*.

*double  
hashing*

We have been discussing hashing for the case where the key is an integer. The same techniques apply if the key is of some other type. For example, the key might be a name. One approach to designing a hash function for such a key would be to assign a number to each letter of the alphabet. (One such function is the function `ord` discussed in the optional section “The Functions `pred`, `succ`, `ord`, and `chr`” of Chapter 8.) The hash function can then add up the numbers associated with each letter of the name to obtain a number and then proceed as if it had a numeric key.

*nonnumeric  
keys*

---

## Variant Records (Optional)

It is frequently convenient and natural to have records in which the field identifiers and component types vary from record to record. For example, a list of publications might naturally be thought of as an array of records. Normally the records would contain

---

```

const MaxI = 100;
type Item =
    record
        Number: integer;
        Price: real;
        Rating: char
    end;
List = array[1..MaxI] of Item;

procedure ShowRecord(R: Item);
begin{ShowRecord}
    writeln('The record contains: ');
    writeln('Item number: ', R.Number);
    writeln('Price: $', R.Price: 6:2);
    writeln('Rating: ', R.Rating)
end; {ShowRecord}

procedure Initialize(var A: List);
{Sets the Number field of each array element equal to 0,
indicating that no record has yet been stored in the position.}
var I: 1..MaxI;
begin{Initialize}
    for I := 1 to MaxI do
        A[I].Number := 0
    end; {Initialize}

function Hash(N: integer): integer;
{Returns an integer in the range 1..MaxI.}
begin{Hash}
    Hash := (N mod MaxI) + 1
end; {Hash}

procedure Insert(var A: List; R: Item);
{Inserts record R into the array A. Precondition: A has been initialized with
the procedure Initialize and after that has been changed only by this procedure.}
var I, Count: 1..MaxI;
begin{Insert}
    I := Hash(R.Number);
    {Look for first available position starting at I.}
    Count := 1; {Counts up to MaxI to detect if the entire array has been inspected.}
    while (A[I].Number <> 0) {position occupied} and
        (A[I].Number <> R.Number) {key not found} and
        (Count < MaxI) {entire array not yet checked} do

```

**Figure 11.14**  
(Optional)  
**Insert and Find**  
procedures using  
hashing.



```

begin{Go to next array position.}
  I := (I mod MaxI) + 1;
  {I := I + 1, but with wrap around at the end so MaxI + 1 is 1.}
  Count := Count + 1
end; {Go to next array position.}
{Either A[I].Number = 0, or A[I].Number = R.Number, or the array is full.}

if A[I].Number = R.Number then
  writeln('Already have an entry for item number ', R.Number)
else if A[I].Number = 0 then
  A[I] := R
else
  writeln('Array is full!')
end; {Insert}

procedure Find(var A: List; Key: integer);
{Searches the array A looking for an index I such that A[I].Number = Key.
Displays the record if it is found; otherwise, announces that it is not in the array.
Precondition: The array A has been initialized with the procedure Initialize
and then changed only by the procedure Insert.}
var I, Count: 1 .. MaxI;
begin{Find}
  I := Hash(Key);
  Count := 1; {Counts up to MaxI to detect if the entire array has been inspected.}
  while (A[I].Number <> Key) {Key not found} and
    (A[I].Number <> 0) {vacant record not found} and
    (Count < MaxI) {entire array not yet checked} do
    begin{Go to next array position.}
      I := (I mod MaxI) + 1;
      {I := I + 1, but with wrap around at the end so MaxI + 1 is 1.}
      Count := Count + 1
    end; {Go to next array position.}
  {Either A[I].Number = Key or no record in the array has its Number field equal to Key.}

  if A[I].Number = Key then
    ShowRecord(A[I])
  else
    writeln('Record is not in the array.')
end; {Find}

```

**Figure 11.14**  
(continued)

slightly different entries for articles and for books. Both articles and books would have an author, a title, and a date. Books normally also have a publisher and city listed, while articles normally list a journal name, volume number, and range of pages. Pascal allows records which have some fields that vary from record to record. These sorts of records are called *variant records*. They are most often used in conjunction with enumerated types. (If you have not read the section on enumerated types, you should go back and read it before continuing with the rest of this section. Enumerated types are covered in Chapter 8.)

*fixed  
part*

The syntax of a variant record type declaration uses something analogous to a *case* statement in order to specify the fields that vary from record to record. For example, Figure 11.15 defines two types. The type `Form` is an enumerated type. The type `Publication` is a variant record type for records, each of which is an entry for either a book or an article. The part before the identifier *case* is called the *fixed part*. It is just like the other record types we have seen. By way of illustration, suppose that we have the following variable declaration:

```
var PubRec: Publication;
```

The following component variables are then exactly like the ones we have seen for the ordinary records discussed in previous sections:

```
PubRec.Author   PubRec.Title   PubRec.Date
```

*variant  
part*

The part of the type declaration that starts with the identifier *case* is called the *variant part*. The identifier `Kind` is called the *tag field identifier*. It is a component of type `Form`. Every record has this component. Hence, `PubRec.Kind` can be used as an ordinary component variable. However, this component has an additional property. The value of the component `Kind` determines the form of the rest of the record. If the value of `Kind` is `Book`, then the record will have the following two additional components:

```
PubRec.Pub
PubRec.City
```

```
type Form = (Book, Article);
  Publication =
    record
      Author, Title: array[1..20] of char;
      Date: 1900..2000;
      case Kind: Form of
        Book:
          (Pub, City: array[1..10] of char);
        Article:
          (Journal: array[1..20] of char;
           Vol, FirstP, LastP: integer)
      end;
```

**Figure 11.15**  
(Optional)  
A variant record  
type.

If the value of `Kind` is `Article`, then these two fields will not exist; instead, the following four fields will be present:

```
PubRec.Journal
PubRec.Vol
PubRec.FirstP
PubRec.LastP
```

In all cases, the types of the components are the ones specified in the declaration.

A complete syntax diagram for record type definitions that can include a variant part is given in Figure 11.16. Notice that there is only one *case* type structure, and it always comes last. (There can be a *case* within the *case*, but such nesting is rare.) All the various field identifiers must be distinct. The `<tag field identifier>` is a normal field identifier. The `<type name>` associated with the `<tag field identifier>` may name any subrange or other ordinal type. It must be a type name; it cannot be a type definition, such as `1 .. 4`.

It is possible to omit the tag field identifier. In Figure 11.15, this would be done by replacing

```
case Kind: Form of
```

with

```
case Form of
```

If this is done, there will be no component called `Kind`. The programmer must somehow ensure that the various cases are used consistently. If a record has its value set using `Pub`, it cannot later try to read the component `FirstP`. If it does, the result is unpredictable. Notice that the tag type must be included even if there is no tag field. The syntax diagram in Figure 11.16 describes all the possible variations.

Variant records are not absolutely necessary. For example, we could have defined the type `Publication` to have all 10 possible fields and could then write the program so that it uses whatever fields are needed. However, on most systems the variant record form will use less storage, because the same storage is used for the `Book` fields as is used for the `Article` fields. Hence, the storage needed is the maximum of the two cases rather than the sum of the two. This kind of storage allocation is illustrated in Figure 11.17.

Figure 11.18 contains a sample program using a variant record type. In that program the tag field `Shape` is set to either `Rectangle` or `Circle`. In the first case, there will be two additional fields called `Height` and `Width`. In the second case, there will be just one additional field, called `Radius`.

*omitting the  
tag field  
identifier*

*storage  
efficiency*

*example*

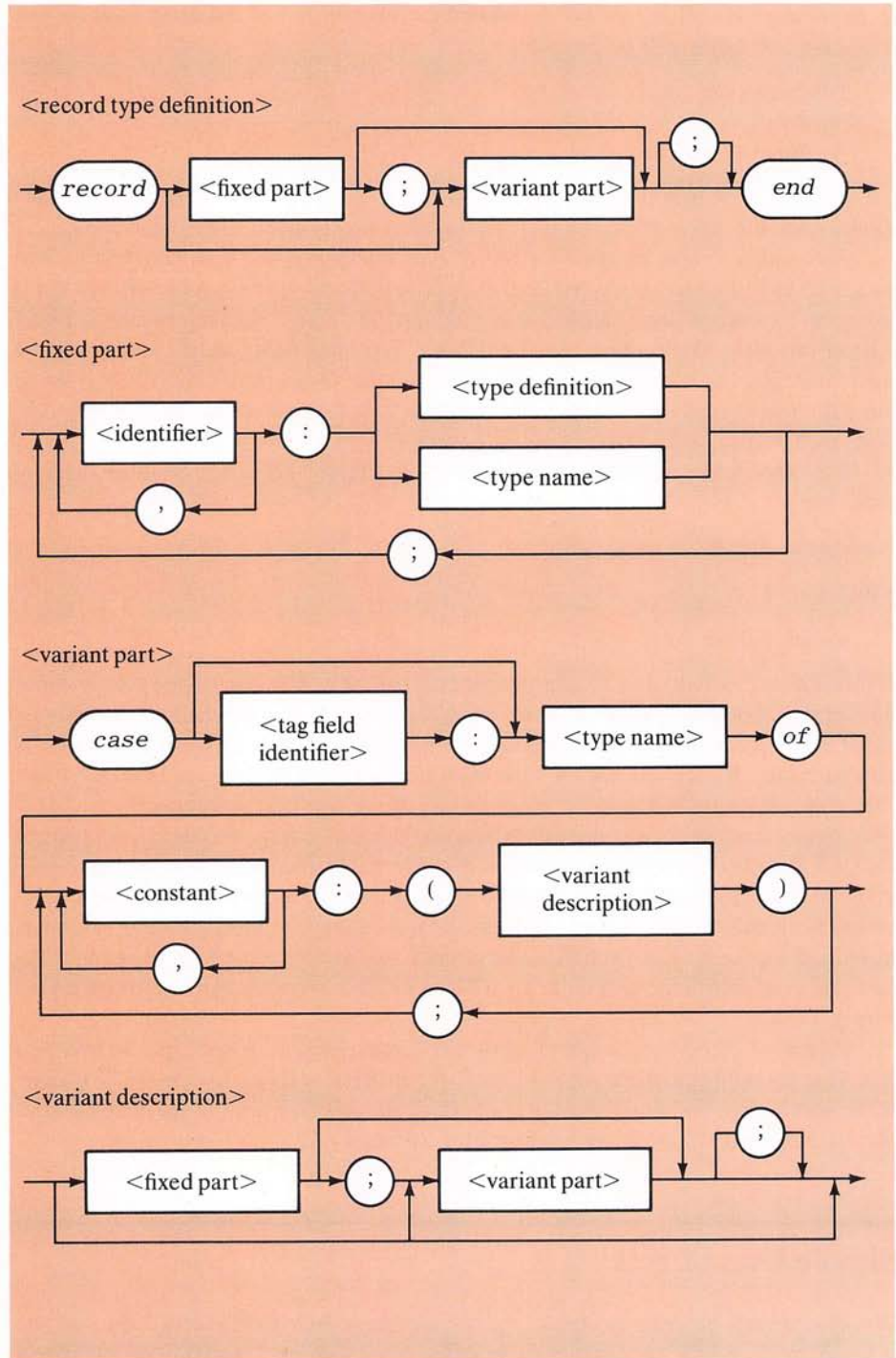
---

## Simple Uses of Sets

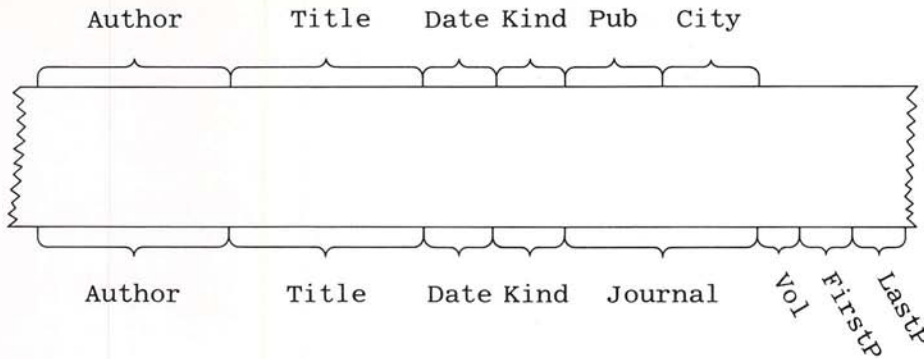
We often want a program to check whether a value is on some list. For example, the program might ask the user to type in the letter `Y` or `N` for “yes” or “no” and then read the answer into the variable `Ans`. But the user might get confused and type in some-

---





**Figure 11.16**  
(Optional)  
Complete syntax  
diagram for record  
type definitions.



**Figure 11.17**  
(Optional)  
**Memory allocation**  
**for a variant**  
**record.**

thing else, such as **OK**. In this situation, you may want the program to make sure the value of **Ans** is either 'Y' or 'N' before going on.

There is a type of boolean expression that can test to see whether the value of an expression is on a specified list of values. The following is a sample of one such expression:

```
Ans in ['Y', 'N']
```

This boolean expression means just what you expect it to mean. It evaluates to **true** if the value of **Ans** is equal to 'Y' or 'N'. In other words, it is exactly equivalent to the boolean expression:

```
(Ans = 'Y') or (Ans = 'N')
```

When the list consists of only two or three values, you can just as well use an expression with *or*, like the one above, but when the values on the list number four or more, this new kind of boolean expression is very handy. For example, the following boolean expressions are completely equivalent, but the first one is easier to write and easier to read:

```
Ans in ['Y', 'y', 'N', 'n']
(Ans = 'Y') or (Ans = 'y') or (Ans = 'N') or (Ans = 'n')
```

The list of values is called a *set*. A set consists of a list of values, separated by commas and enclosed in square brackets. Only square brackets can be used. Curly brackets { } are not allowed.

To form one of these boolean expressions, you write an expression to be evaluated followed by the reserved word *in*, followed by a set. For example,

```
(X + Y) in [8, 34, 17]
```

The type of the expression in front of the *in* must evaluate to a value of an ordinal type, such as integer or char. Notice that this means the expression cannot be of

**Program**

```

program AreaComp(input, output);
{Illustrates the use of variant records by computing
the area of either a rectangle or a circle.}
const Pi = 3.14159;
type Kind = (Rectangle, Circle);
  Figure =
    record
      Area: real;
      case Shape: Kind of
        Rectangle: (Height, Width: real);
        Circle: (Radius: real)
      end;
var F: Figure;
    Ans: char;

function Convert(Letter: char): Kind;
{Converts a letter code into a value of type Kind.
Upper- or lowercase 'R' for Rectangle, 'C' for Circle.}
begin{Convert}
  case Letter of
    'C', 'c': Convert := Circle;
    'R', 'r': Convert := Rectangle
  end {case}
end; {Convert}

begin{Program}
  {Determine the kind of figure.}
  writeln('Enter type of figure:');
  writeln(' (R for Rectangle, C for Circle) ');
  readln(Ans);
  F.Shape := Convert(Ans);

  {Obtain the necessary data and compute the Area.}
  case F.Shape of
    Circle: begin{Circle}
      writeln('Enter radius:');
      readln(F.Radius);
      F.Area := Pi * sqr(F.Radius)
    end; {Circle}
    Rectangle: begin{Rectangle}
      writeln('Enter height and width:');
      readln(F.Height, F.Width);
      F.Area := F.Height * F.Width
    end {Rectangle}
  end; {case}
end; {Program}

```

**Figure 11.18**  
(Optional)  
Use of variant  
records.



```
writeln('Area = ', F.Area);
case F.Shape of
  Circle: writeln('For radius = ', F.Radius);
  Rectangle: writeln('For dimensions ',
                    F.Height, ' by ', F.Width)
end {case}
end. {Program}
```

### Sample Dialogue 1

```
Enter type of figure:
(R for Rectangle, C for Circle)
R
Enter height and width:
1 2
Area = 2.0
For dimensions 1.0 by 2.0
```

### Sample Dialogue 2

```
Enter type of figure:
(R for Rectangle, C for Circle)
C
Enter radius:
2
Area = 12.5663
For radius = 2.0
```

**Figure 11.18**  
(continued)

type real. All the constants in the list must be of the same type as the expression. The entire boolean expression evaluates to *true* if the value of the expression, such as  $(X + Y)$ , is on the list; otherwise, it evaluates to *false*.

Since these expressions involving *in* are boolean expressions, they may be used as subparts of other, more complex boolean expressions containing *and*, *or*, or *not*.

The syntax of how *not* and *in* are combined does not mirror English usage, and this sometimes causes confusion. Note that the following is not allowed as a Pascal boolean expression:

```
Ans not in ['Y', 'N']
{NOT ALLOWED IN PASCAL}
```

The correct syntax is

```
not(Ans in ['Y', 'N'])
```

## TURBO Pascal Pitfall

### Limitations on Sets

The items on the list in a set are called the *elements* of the set. For example, 'Y' and 'N' are the elements of the set ['Y', 'N'], and the numbers 2, 3, and 5 are the elements of the set [2, 3, 5].

In TURBO Pascal there is a limitation on the integers that are allowed as elements of sets. Only the integers from 0 to 255 may be used in TURBO Pascal sets. Both 0 and 255 may be used, but -1 and 256, as well as all other numbers outside the range 0 . . . 255, may not be used. Hence, the following expression is not allowed in TURBO Pascal:

(X + Y) in [8, 34, -30]

This expression will not work because the set includes the element -30, which is outside the allowable range.

The reason for this limitation has to do with the way that sets are implemented in TURBO Pascal. The restriction allows sets to be represented in a simpler way in the computer's memory. It was done to make the compiler's job easier and to make the programs run faster. It has no obvious conceptual explanation.

There is no restriction on which characters can be used. Any letter may be used in a set expression. However, do not forget that you cannot mix letters and integers in a Pascal set expression. All the elements in a set expression must be of the same ordinal type.

Figure 11.19 gives an innocent looking statement using a set expression. Although innocent looking and, moreover, allowed in most standard Pascal implementations, it will not work in TURBO Pascal. In TURBO Pascal the alternative expression using *or* must be used instead.

### More about Sets

(Optional)

*sets  
versus  
lists*

A *set* is almost identical to what most people call a "list." However, there are some differences. The elements of a list have an order, and that order is part of the identity of the list. Thus, the following two lists are different:

1, 3, 5  
1, 5, 3

A set is like a list except that the order in which the elements are named is unimportant. For example, the following two Pascal sets are equal:

[1, 3, 5]  
[1, 5, 3]

```
{THE FOLLOWING IS NOT ALLOWED IN TURBO PASCAL:}
if X in [2, 59, 307] then
    writeln('X is 2, 59 or 307')

{THE FOLLOWING WORKS CORRECTLY IN TURBO PASCAL:}
if (X = 2) or (X = 59) or (X = 307) then
    writeln('X is 2, 59 or 307')
```

**Figure 11.19**  
**Set restrictions in**  
**TURBO Pascal.**

One other difference between sets and lists is that unlike a list, a set cannot have any repetitions.

In the previous section, we used set constants like ['Y', 'N']. Pascal also allows variables whose values are sets. These set variables are declared to be of one of the defined types known as *set types*. Set types are structured types like arrays and records. The values of a set type are built up from a group of values of some ordinal type. In the case of sets, these simple values which together form the set are called the *elements* of the set. For example, the set ['Y', 'N'] has two elements, 'Y' and 'N'. In Pascal, the elements of a set must all be of the same type, and that type must be an ordinal type. Thus, a set can contain integers only or characters only, but not a combination of integers and characters. A set type definition takes the following form:

```
set of <base type>
```

The <base type> is an ordinal type, such as char or a subrange type. Set types are declared along with all the other defined types, such as arrays and records. Once a set type has been declared, a program can declare variables to be of a set type in the usual ways. For example,

```
type SmallSet = set of 0..10;
var Scores, CheckOn: SmallSet;
    Symbols: set of char;
```

Once a variable is declared, it can have its value changed by an assignment statement, such as

```
Scores := [1, 3, 5]
```

Expressions such as [1, 3, 5] are constants of a set type. As we have already seen, a *set constant* can be written by giving a list of elements of the base type separated by commas and enclosed in square brackets. If some of the elements of a set constant form a subrange, then it is possible to abbreviate the list of elements by listing a subrange type. For example,

```
CheckOn := [0..5, 10]
```

This statement sets the value of CheckOn equal to

```
[0, 1, 2, 3, 4, 5, 10]
```

*set*  
*types*

*elements*

*set*  
*constants*



*set  
operators*

Sets can be manipulated with the operators given in Figure 11.20 and can be compared using the relational operators given in Figure 11.21. For example, the following boolean expressions all evaluate to true:

```
['A', 'B', 'C'] = ['B', 'C', 'A']
['A', 'B'] <> ['A', 'B', 'C']
[1, 2] <> [5, 7, 22]
[1, 2] <= [1, 2, 3]
[1, 2] <= [1, 2]
[1, 2, 3] >= [1, 2]
[1, 3] + [3, 5, 7] = [1, 3, 5, 7]
[1, 3] * [3, 5, 7] = [3]
[1, 2, 3, 4] - [4, 2, 5] = [1, 3]
```

**Figure 11.20**  
(Optional)  
Set operators.

Pascal Form	Definition of Value Returned
<set 1> + <set 2>	Union: the set containing all the elements that are in <set 1> or <set 2> or both
<set 1> * <set 2>	Intersection: the set containing those elements that are in both <set 1> and <set 2>
<set 1> - <set 2>	Set difference: the set containing those elements that are in <set 1> but not in <set 2>

**Figure 11.21**  
(Optional)  
Set relational  
operators.

Pascal Form	Definition of Value Returned
<set 1> = <set 2>	Equality: evaluates to true if <set 1> and <set 2> contain exactly the same elements; otherwise, it evaluates to false
<set 1> <> <set 2>	Inequality: evaluates to true if <set 1> and <set 2> are not equal; otherwise, it evaluates to false
<set 1> <= <set 2>	Subset: evaluates to true if every element of <set 1> is also an element of <set 2>; otherwise, it evaluates to false
<set 1> >= <set 2>	Superset: evaluates to <set 2> <= <set 1>

The operator *in*, as well as the operators in Figures 11.20 and 11.21, can be applied to set variables as well as to set constants.

Figure 11.22 contains a sample program using set variables. Notice that `[ ]` is used to denote the set with no elements in it. As you might easily guess, that set is called the *empty set*. Also notice that a variable (or other expression) of the base type can be used within a set expression, as in the following line, which appears in the procedure `ReadSentence`:

*empty  
set*

```
ChSet := ChSet + [Character];
```

### Program

```
program Letters(input, output);
{Requests a sentence from the keyboard, forms a set consisting of all symbols
in the sentence, writes out a list of all the uppercase letters used in the sentence.}
type SetOfChar = set of char; {Procedure declarations
                                require named types. Hence, this silly-looking type definition.}
var ChSet: SetOfChar;

procedure ReadSentence(var ChSet: SetOfChar);
{Requests a sentence from the keyboard, then reads the sentence
and sets the value of ChSet equal to the set of all characters in the
sentence. Sentence must terminate with a period, question mark,
or exclamation point. The terminator is not put in the set ChSet.}
var Character: char;
    Terminators: SetOfChar;
begin{ReadSentence}
    writeln('Enter a sentence. End it with');
    writeln('a '?'', '!', or a period. ');

    Terminators := [ '.', '?', '!'];

    ChSet := [];
    read(Character);
    while not (Character in Terminators) do
        begin{while}
            ChSet := ChSet + [Character];
            {ChSet contains the characters read so far.}
            read(Character)
        end {while}
    end; {ReadSentence}
```

**Figure 11.22**  
**(Optional)**  
**Program using set**  
**variables.**

```

procedure WriteLetters(ChSet: SetOfChar);
{Writes out all the uppercase letters that are in the set CharSet.
Each letter is written out only once. (On systems that have characters other
than uppercase letters in the range 'A' . . 'Z', those symbols will also be
written out if they are in the set CharSet.)}
const Blank = ' ';
var Character: char;
begin{WriteLetters}
  for Character := 'A' to 'Z' do
    if Character in ChSet then
      write(Character, Blank);
  writeln
end; {WriteLetters}

begin{Program}
  ReadSentence(ChSet);
  writeln('Your sentence contains the');
  writeln('following uppercase letters:');
  WriteLetters(ChSet)
end. {Program}

```

#### Sample Dialogue

Enter a sentence. End it with  
a '?', '!', or a period.  
**Buy Low and Sell HIGH!**  
Your sentence contains the  
following uppercase letters:  
B G H I L S

**Figure 11.22**  
(continued)

## TURBO Pascal

### Defined Set Constants

(Optional)

TURBO Pascal allows typed constants of set types. The syntax is similar to that of other typed constants and is illustrated by the following examples:

```

type Grade = set of 0 .. 10;
const Pass: Grade = [6 .. 10];
type SetOfChar = set of char;
const Vowels: SetOfChar = ['a', 'e', 'i', 'o', 'u'];

```



Hence, in TURBO Pascal we could rewrite the program in Figure 11.22 so that the variable Terminators is a typed constant rather than a variable. The required declarations are

```
type SetOfChar = set of char;  
const Terminators: SetOfChar = ['.', '?', '!'];
```

---

Algorithms + Data Structures = Programs  
*Niklaus Wirth (book title)*

---

---

## Summary of Problem Solving and Programming Techniques

- Records can be used to combine data of different types into a single compound value of a record type.
  - A component variable of a record variable can be used anyplace that a simple variable of the component type can be used.
  - A record variable without a field identifier can be used in an assignment statement, like so: `Item1 := Item2`.
  - Either a record component or an entire record may be passed as a parameter to a procedure. In the first case, the formal parameter type must be a component type of the record; in the second case, it must be the record type.
  - Hierarchical control structures, like hierarchical data structures, make a program easier to understand. It pays to combine the basic data structure types to obtain hierarchical structures such as arrays of records, records of arrays, arrays of records of arrays, and so forth.
  - When you are manipulating arrays of records (and other hierarchical data structures), it simplifies notation and reasoning to use procedures that have a record value (or variable) as a parameter or parameters, and to manipulate the records with procedures. This is a form of data and procedural abstraction that eliminates notational detail from the body of the program.
  - Always consider the possibility of alternative data structures. Just because you find one that works does not mean that you have found the best one.
  - Parallel arrays and arrays of records are data structures that serve the same function. Which one you choose for a particular application will depend on the details of that application.
-

## Summary of Pascal Constructs

### simple record type declaration

Syntax:

```
type <type name> =  
    record  
        <field identifier 1> : <component type 1>;  
        <field identifier 2> : <component type 2>;  
        .  
        .  
        .  
        <field identifier n> : <component type n>  
    end;
```

Examples:

```
type Person =  
    record  
        Name: array[1 . . 20] of char;  
        Age: 1 . . 100;  
        Height: real;  
        Weight: real  
    end;  
Sample =  
    record  
        A: integer;  
        B: char  
    end;
```

The <type name> is an identifier that will name the record type. The field identifiers <field identifier 1>, <field identifier 2>, . . . , <field identifier n> can be any non-reserved-word identifiers. All these identifiers must be different. The component types may be any Pascal types and may be different from one another. If two field identifiers are of the same type, then their declaration may be combined by separating them with commas and only listing the component type once, like so:

```
type Person =  
    record  
        Name: array[1 . . 20] of char;  
        Age: 1 . . 100;  
        Height, Weight: real  
    end;
```

The above type declaration is equivalent to the first type declaration given in the examples.

---

**record variable declaration**

Syntax:

```
var <variable name>: <type>;
```

Example:

```
var MsX, MrY: Person;
    Sam: Sample;
```

This form is the same as any other variable declaration. Note that no field identifiers are used.

**component variable of a record variable**

Syntax:

```
<record variable> . <field identifier>
```

Example:

```
MsX.Height
```

This is a variable of the type given after <field identifier> in the type definition for the type of the <record variable>.

**variant record (optional)**

Syntax:

```
type <type name> =
    record
        <field identifier 1> : <component type 1>;
        <field identifier 2> : <component type 2>;
        .
        .
        .
        <field identifier n> : <component type n>
    case <tag field identifier> : <type name> of
        <label 1> : (<variant description 1>);
        <label 2> : (<variant description 2>);
        .
        .
        .
        <label n> : (<variant description n>)
    end;
```

} Fixed  
part

} Variant  
part



Example:

```
type Kind = (Rectangle, Circle);
Figure =
  record
    Count: integer;
    case Shape: Kind of
      Rectangle: (Height, Width: real);
      Circle: (Radius: real)
    end;
```

The fixed part is the same as in a simple record type definition, which is described in the first entry of this summary. The <tag field identifier> is any non-reserved-word identifier. The <type name> that follows is the type name of a subrange or ordinal type. It is the type for the field named <tag field identifier>. The <label *i*> are each labels, or a list of labels, of this type. Each <variant description *i*> is a list of field identifiers followed by their types, with colons and semicolons inserted as in the fixed part. (A <variant description *i*> can also contain a variant part of its own, although this sort of nesting is rare.) All the field identifiers must be different.

#### **with statement**

Syntax:

```
with <record variable list> do
  <statement>
```

Example:

```
with MsX, Sam do
  begin
    Height := 5.2;
    A := 8
  end
```

The <record variable list> is a list of record variables with no field names in common. Within the <statement>, the component variables of the record variables on the list may be referred to by using only the field identifier. For example, given the declarations in the above entries, the preceding *with* statement is equivalent to the following compound statement:

```
begin
  MsX.Height := 5.2;
  Sam.A := 8
end
```

#### **use of in**

Syntax:

```
<expression> in <set>
```

---

Examples:

```
X + Y in [3, 4, 7]
Grade in ['A', 'B', 'C']
```

A kind of boolean expression. The *<expression>* before the *in* is an expression of an ordinal or subrange type. *<set>* is a list of constants separated by commas and enclosed in square brackets. The constants must all be of the same type as *<expression>*. This boolean expression evaluates to *true* if the value of *<expression>* is equal to the value of one of the constants on the list.

### type declaration for set types (optional)

Syntax:

```
type <type name> = set of <base type>;
```

Examples:

```
type SymbolSet = set of char;
GradeSet = set of 0 .. 10;
```

The way to declare a set type. The *<base type>* may be any ordinal or subrange type. However, most implementations require that the size of the *<base type>* be relatively small. The type *integer* is too large. The type *char* is allowed on most, but not all, systems. There may be variables of a set type, and the values may be combined using the operations given in Figures 11.20 and 11.21.

---

## Exercises

### Self-Test Exercises

5. Suppose that *B* is declared as follows, where the type is as declared in Figure 11.13.

```
var B: EntryList;
```

Write Pascal expressions for each of the following: the third item on the list *B*; the price of the third item on the list *B*; the sixth letter of the name of the tenth item on the list; the length of the name of the second item on the list.

6. The inventory of a shoe store lists shoes by a stock number. With each stock number there is associated a style number in the range 0 to 50, the number of pairs in each size (sizes range from 3 to 14), and a price. A program is to be written to keep track of the inventory. Give type declarations for two different ways to structure the inventory data: as parallel arrays and as an array of records.

7. (This exercise uses the optional section “More about Sets.”) Determine the value returned by each of the following expressions:

```
[7, 8, 9] + [8, 1, 3]
[7, 8, 9] + []
```

---

```

[7, 8, 9] * [8, 1, 3]
[7, 8, 9] * [31, 19]
[7, 8, 9] - [8, 9, 15]
[7, 8, 9] = [8, 9, 7]
[7, 8, 9] = [1, 2, 3]
[7, 8, 9] <> [1, 2, 3]
[9, 8, 5] >= [8, 9]
['A', 'C', 'D'] <= ['A', 'C']
['A', 'B', 'C'] >= ['B', 'A']

```

### Interactive Exercises

8. Write a program to read data from the keyboard into a record of the type given below and to then echo the data back to the screen.

```

type Sample = record
    A: char;
    B: integer
end;

```

9. Redo the previous exercise, but this time replace the type `char` with the type `CharString` declared in Figure 11.11.

10. Write a program to fill (with data read from the keyboard) one record of the type `Entry` declared in Figure 11.13. Use a *with* statement.

### Programming Exercises

11. Write a program that records the inventory for a shoe store in an array of records. Use the type declaration from Exercise 6. (See “Answers to Self-Test Exercises” in the back of the book.) The user is given the following choices: enter a new record, display a record, change the price of a stock item, or change the number on hand. When specifying a record, the user may give either the stock number or the style number. The array index can be used as a stock number. If the user decides to change the stock on hand, the program should ask which sizes will have their stock on hand changed. The program should be designed to run indefinitely, keeping track of changes in stock.

12. Write an inventory program for items whose record descriptions are of type `Entry`, given in Figure 11.13. The program reads in a list of records entered at the keyboard and stores them in an array of records. The program then allows the user to inquire about records. The user can ask to see all records with a given field name; all records in a given price range, such as \$5.00 to \$9.99; or all records of a given quantity range, such as over 100, or under 2, or between 10 and 20.

13. Write a grading program for a class with the following grading policies:

- a. There are three quizzes, each graded on the basis of 10 points.
- b. There is one midterm exam and one final exam, each graded on the basis of 100 points.



- c. The final exam counts for 50% of the grade, the midterm counts for 25%, and the quizzes count for 25%. (Do not forget to normalize the quiz scores. They should be converted to a percentage before they are averaged in.) Grades are determined by the following rule: 90 . . . 100 is an A, 80 . . . 89 is a B, 70 . . . 79 is a C, 60 . . . 69 is a D, and below 60 is an F.

The program reads in the student's name and various scores and outputs a table of students showing the student's name, all scores, average numeric score, and final letter grade. The program also outputs the class average for the final numeric score.

14. Modify the previous exercise in the following ways:

- The table of students also shows the difference between each student's final numeric score and the class average.
- The program also calculates the median value of the final numeric scores. (The median score is the score such that there are as many above it as below it, that is, the "midpoint" score. If there is an even number of scores, there are two scores in the middle. In that case, the median is the average of those two scores.)
- The table of students also shows the difference between each student's final numeric score and the median score.

15. Write a program that reads a line of text into an array of records of type `CharString`, as defined in Figure 11.11. Each word is read into one record of the array. Each punctuation symbol, such as a comma, colon, and so forth, is also read into a single record. (You may assume that the only punctuation symbols are commas, colons, periods, exclamation points, and question marks.) The sentence is corrected for spaces and capitalization and then output. For example, the input

hi , How are you?

should produce the output

Hi, how are you?

You need not be concerned about capitalizing proper names. If "John" is changed to "john," that will be acceptable. You will want to write a procedure `StringWrite` that is like `StringWriteLn` except that it does not advance to the next line. (The code is the same except that the final `writeln` is omitted.) You will probably want to design a special procedure for reading single words, rather than using the procedure `StringReadLn`.

16. Write string-manipulating procedures for strings of the type `CharString`, as declared in Figure 11.11. There should be a concatenation function; for example, the concatenation of "do be" and "do" is "do bedo". There should be a pattern-searching function like the function `Subpattern` in Figure 10.12 but applicable to this new definition of the type `CharString`. There should also be a procedure to delete a substring specified by symbol positions; for example, the procedure should be capable of deleting symbols number two through four from "abcdefg" to obtain "aefg". The string being manipulated should be a variable parameter that is changed by the proce-

dure. Further, there should be a procedure to insert a string at a given position; for example, the procedure should be capable of accepting instructions equivalent to *insert "sam" after location 2 in "dobedo"* to deliver the string "dosambedo". The string like "dobedo" is a variable parameter whose value is changed to the new value like "dosambedo". Embed these procedures in a test program.

17. (You should know about random number generators to do this exercise. They are covered in Chapter 8.) Write a procedure declaration for a procedure called *Deal* that sets the values of a variable parameter of a record type to values that represent a card chosen at random from a standard 52-card deck. The function should also keep track of the cards already dealt out, so that it does not deal a card twice. A card will be represented as two component values, one for the "value"—ace, two, and so forth—and one for the suit—diamonds, clubs, and so forth. Use an array-of-records parameter to keep track of the cards already dealt out.

18. (You should know about random numbers to do this exercise. They are covered in Chapter 8.) Write a program to play "clock patience," displaying the game configurations on the screen. *Clock patience* is a solitaire card game played as follows: The cards in a 52-card deck are dealt into 12 piles of 4 each in a "clock" circle, with the remaining 4 cards in the middle. A move consists of taking a card from a pile and placing it under the pile where it belongs, and this pile provides the card for the next move. (Cards are ordered clockwise: ace for one, then two, three, and so forth, up to queen.) The center pile is for the kings. The game terminates when the four kings have been placed on the center pile. The game is considered successful if all the other cards are correctly placed.

19. Write a program to score five-card poker hands into one of the following categories: nothing, one pair, two pairs, three of a kind, straight (in order), flush (all the same suit), full house (one pair and three of a kind), four of a kind, and straight-flush (both straight and flush). Use an array of records to store the hand. The array index type is 1 . . . 5, and the records have one field for the value and one for the suit of a card.

20. (You should know about random number generators to do this exercise. They are covered in Chapter 8. Exercises 17 and 19 should be done before you do this one.) Write a program to play five-card draw poker with the user. The user and the program each get five cards. They may discard and receive replacements for up to three cards. The hands are then scored according to the order given in the previous exercise. Do not forget to keep track of the cards already dealt so that no card is dealt twice. In the easy version, only the above ordering is used, and so any two hands with three of a kind, for example, are equal. In the more difficult version, the hands are compared further; for example, three aces beats three jacks.

---

## References for Further Reading

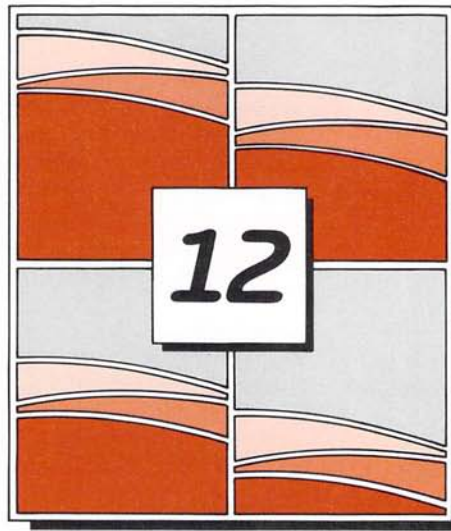
A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, 1983, Addison-Wesley, Reading, Mass.

---

- 
- C.A.R. Hoare, "Notes on Data Structuring," in O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, 1972, Academic Press, New York.
- D.F. Stubbs and N.W. Webre, *Data Structures with Abstract Data Types and Pascal*, 1985, Brooks/Cole Publishing, Monterey, Ca.
- A.M. Tenenbaum and M.J. Augenstein, *Data Structures Using Pascal* 2nd Edition, 1986, Prentice-Hall, Englewood Cliffs, N. J.
-







## *Program Design Methodology*

The Analytical Engine has no pretensions whatever to *originate* anything. It can do whatever we *know how to order it* to perform. It can *follow* analysis; but it has no power of *anticipating* any analytical relations or truths. Its province is to assist us in making *available* what we are already acquainted with.

*Ada Augusta, Countess of Lovelace*

## Chapter Contents

Some Guidelines for Designing	Efficiency versus Clarity
Algorithms	Off-Line Data
Writing Code	Coping with Large Programming
Data and Procedural Abstraction	Tasks
Testing and Debugging	Summary of Terms
Verification	Exercises
Portability	References for Further Reading
Efficiency	

**T**he production of a program can be divided into two phases: the problem solving phase, including problem definition and algorithm design; and the implementation phase, during which the actual program code is produced. Testing and debugging take place during both of these phases. Throughout this book we have described a number of techniques to apply at various points in this process. In this chapter we summarize these techniques and then go on to discuss a few other issues connected with the design and maintenance of computer programs.



---

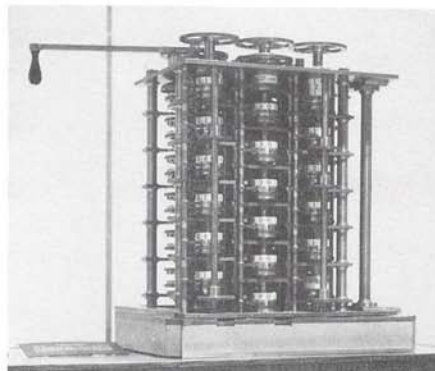
## Some Guidelines for Designing Algorithms

The first computer that was similar in character to today's machines was designed by Charles Babbage, an English mathematician and physical scientist. The project began sometime before 1822, consumed the rest of his life, and, although he never completed the construction of the machine, the design was a conceptual milestone in the history of computing. His colleague, Ada Augusta, was an interesting figure in a number of ways. She was the daughter of the poet Byron. Later she became Countess of Lovelace. It is primarily through her writings that the work of Babbage has been made available to the world. Indeed, she is frequently given the title of the first computer programmer. Her comments, quoted in the chapter opening, still apply to the process of solving problems on a computer. Computers are not magic and do not, at least as yet, have the ability to formulate sophisticated solutions to all the problems we encounter. Computers simply do what the programmer *orders* them to do. The solutions to problems are carried out by the computer, but the solutions are formulated by the programmer. Hence, in order to solve problems on a computer, you must know how to design algorithms.



Ada Augusta,  
Countess of  
Lovelace and the  
first computer  
programmer,  
left.

Charles Babbage,  
right.



A model of  
Babbage's  
computer

---

*defining  
the problem*

If we could provide you with a method that was guaranteed to lead you to a correct algorithm for any problem you might encounter, then programming would be a very simple task. However, neither this nor any other book can provide such a method. There is no algorithm for writing algorithms. Designing algorithms is a creative process. There are, however, some guidelines that can sometimes help in your search for algorithms. These guidelines have some similarity to an algorithm for writing algorithms. They do, however, fall short of being an algorithm for producing algorithms in two ways: The steps are not precisely defined, and they are not guaranteed to produce a correct algorithm. The guidelines are listed in Figure 12.1.

The original formulation of a problem is typically imprecise, incomplete, or both. Before attempting to formulate an algorithm for a problem, be sure that the problem definition has been made complete and precise. In particular, be certain that

- You have complete and precise specifications for the inputs that will be needed.
- You have complete and precise specifications for the output.
- You know how the program must react to incorrect data: Is it required to issue an error message? Must it be able to continue computing, or may it end when it encounters input of the wrong type or in the wrong format?
- You know when the program should end and how the program will know when it is time to end.

#### Some Guidelines for Algorithm Design

1. Formulate a precise statement of the problem to be solved by the algorithm.
2. See if somebody has already formulated an algorithm to solve the problem.
3. See if any standard techniques (“tricks”) can be used to solve the problem.
4. See if the problem is a slight variation of a problem for which somebody has already formulated an algorithm. If so, try to adapt that known algorithm to the new problem.
5. Design a data structure to organize the data involved.
6. Break the problem into subproblems and apply this method to each of the subproblems.
7. If all else fails, simplify the problem, and apply this method to the simplified problem. When you obtain a solution to the simplified problem, try to adapt the algorithm to fit the real problem. If that fails, or if the algorithm produced is unclear, incorrect, or inappropriate to the real problem, then discard this first attempt and start the process all over again at step 1. (You should then have a better feel for the problem and a better chance of success.)

**Figure 12.1**  
**Guidelines.**



The algorithm may be produced at any point after step 1. The earlier the better. For example, if you are asked to write a program for computing the square root of a number, you could design an algorithm from scratch. However, if you simply take an existing algorithm that has stood the test of time, it is likely to be more efficient, and it is less likely to contain any subtle errors. In this extreme case, there is even a predefined Pascal function to do the task. In other cases, such as sorting a list, there is no predefined Pascal procedure for the task, but there are a number of well-known algorithms for the problem, such as the sorting algorithm discussed in Chapter 9. You may wish to solve these problems on your own as a training exercise, but in a “real world” situation, where it is the performance of the program that counts, you should always see what algorithms others have produced.

*using known  
algorithms*

Step 3 requires that you cultivate a “bag of tricks,” or do a search of the literature, or both. A number of tricks are well known to experienced programmers. In Chapter 7 we gave a number of well-known techniques for terminating a loop that reads in input data. An example of one of the tricks we presented there was the use of a sentinel value to mark the end of a list of input numbers. Other, more complicated tricks are discussed throughout this book. Frequently, these so-called “tricks” are rare and brilliant insights that are easy to understand and use but difficult to discover on your own.

*standard  
tricks*

As a simple example of step 4, consider the following code for computing the sum of the numbers stored in an array A:

```
Sum := 0;
for I := First to Last do
    Sum := Sum + A[I]
```

*adapting  
another  
algorithm*

If we instead wish to compute the product of all the numbers in the array, we can adapt the algorithm by substituting multiplication for addition and by substituting 1 for 0. If we also rename the variable Sum to Product, that yields:

```
Product := 1;
for I := First to Last do
    Product := Product * A[I]
```

A more complicated example is given in Chapter 11, where we take an algorithm for sorting arrays of numbers and adapt it to obtain a procedure to sort arrays of records.

If there is no existing algorithm or standard technique that can be applied to the problem, then you must design an algorithm from scratch. But before going on to design the algorithm, first design a suitable data structure. Steps 5 and 6 are not unrelated. The choice of a data structure will influence the algorithm you design and vice versa. As you design an algorithm, it may prove convenient to go back and change the data structure. Do not feel that you are irrevocably committed to a data structure, and do not feel that all the details of the data structure must be determined before you begin the algorithm design, but you should have some data structure in mind when you are designing your algorithm. Chapters 9, 10, and 11 discuss the design of data structures composed of arrays and/or records. Other data structures are discussed in Chapters 13, 16, and 17.

*choosing a  
data  
structure*

Step 6 is the top-down design strategy that we have been using and advocating throughout this book. The subproblems are attacked by these same guidelines, starting

*top-down  
design*



with step 1. Eventually the subproblems become so small that their solution is obvious or one of the other steps applies, such as when there is a well-known solution to the subproblem.

*when  
all else  
fails*

Sometimes step 7 can be a variation on step 6. First design an algorithm with some features missing, and then design embellishments for the missing features. The change-making program we designed earlier in this book is an example of this technique. In Chapter 4 we designed a basic program for calculating change. In Chapter 6 we enhanced it to have much neater and clearer output.

Step 7 is also the step to use when you are completely stumped by a problem and need a way to overcome a mental block. If you cannot solve the problem at hand, solve a related and simpler problem as a practice exercise. That should give you some new insights. Then throw out the practice algorithm and start over. Do not be reluctant to throw out an algorithm or program. It is usually faster and easier to design an algorithm from scratch than to salvage a poor design.

These are all just guidelines. You should always consider them, but you need not adhere to them rigidly. In particular, the order of steps (especially the order of steps 3 through 6) is certainly not rigid. Moreover, they are certainly not a complete list of known techniques. They are merely a general plan of attack that can and should be augmented with other design techniques. In Chapter 7 (particularly in the optional sections), we discussed some additional techniques that apply to the design of loops. Those techniques can be used in conjunction with the plan of attack given in Figure 12.1.

---

## Writing Code

When a program is divided into subtasks, the algorithms for the subtasks can and should be coded and tested separately. Even when you are simply coding somebody else's algorithm, divide the algorithm into subparts and code the subparts separately. That way they can be tested and debugged separately. It is relatively easy to find a mistake in a small procedure. It is nearly impossible to find all the mistakes in a large, untested program that was coded and tested as a single large unit rather than as a collection of well-defined modules. The rules we have been advocating for indenting and documentation all apply to the task of writing code. In fact, they apply even earlier. Pseudocode should have an indenting pattern and a collection of comments that will carry over with only minor changes to the final Pascal code.

---

## Data and Procedural Abstraction

Programs should have a conspicuously modular design. The program should be divided into self-contained subpieces. These pieces are usually implemented as procedures and are a natural consequence of the top-down design strategy we have been advocating. Each procedure should be a self-contained piece that is defined by a brief comment explaining what it does. After the procedure is written and tested, this comment should be all you need to know in order to use the procedure. If the procedure is designed with

---

this in mind, then the details of how the procedure performs its task can safely be forgotten. This form of selective forgetfulness is called *procedural abstraction*. It keeps our reasoning free of the clutter of small detail. Using procedural abstraction, we can build up a library of procedures whose internal structure may be very complicated but which are, nonetheless, straightforward to use. For example, in order to use the procedure `Sort` in Figure 9.16, you do not need to go back and read the code to find out how it does the sorting. All you need to know is that it does somehow sort an array of integers.

This same method of abstraction can be applied to data structures as well as to procedures. A data structure often has some incidental detail that is required by the rules of the language but that has nothing to do with the problem at hand. For example, an array to hold 10 numbers might be indexed by the subrange type `1 .. 10` or `0 .. 9` or even some other range of 10 numbers. Typically, the particular index type used is of no relevance to the problem at hand. So, rather than remembering this irrelevant detail, it is clearer to use the type `Low .. High` where the values `Low` and `High` are defined constants whose values we normally do not “remember.” A more sophisticated example of data abstraction can be found in the discussion of the record type `CharString` designed in Chapter 11.

---

## Testing and Debugging

When you are producing a program, a natural sequence of events is algorithm design, coding, and then testing and debugging. However, the divisions are not, or at least should not be, very rigid. Some coding and debugging for certain tasks can be done before the complete algorithm is derived. When a task is broken into subtasks, some simple testing can and usually should be done immediately. If the tasks are well defined and fit together simply enough, then a pencil-and-paper simulation of the algorithm can be used to test this formulation of the problem. If the test cannot be carried out with pencil and paper, then a skeleton of a program can be written to see whether the pieces will fit together. If the pieces do not fit together, then there is no point in proceeding to derive algorithms for the subtasks.

For example, consider the following task: Suppose we wish to gather statistics on how much time student programmers spend at the terminal during any one session. Perhaps this will be used to help redesign the chairs so that students will be more comfortable while coding their homework assignments. To get a good profile of usage, we want a program that will compute the average time per session for each individual student, as well as the average session length averaged over all sessions by all students. The input is to consist of a list of students, with each student's name followed by a list of the times that student spent in each individual session. Dividing this task into subtasks might produce the following pseudocode:

*example  
of early  
testing*

1. Compute each student's average time per session.
  2. Output the averages.
  3. Compute the average of the student averages obtained in 1 and output that as the overall average.
-



The presence of a logical flaw in this algorithm can be discovered by testing, even before algorithms for the subtasks are designed. Consider the following sample data:

Joseph Cool:

10 min., 10 min.

Sally Workhard:

60 min., 60 min., 60 min., 60 min.,

60 min., 60 min., 60 min., 60 min.

We do not need a computer to simulate the pseudocode on this simple data. The two student averages are trivially seen to be 10 and 60 min. Now step 3 of the pseudocode says to average these two numbers, which yields

$$(10 + 60)/2 = 35 \text{ min.}$$

The algorithm's computation of the average session length is wrong. A little bit of pencil and paper, or mental arithmetic or a hand calculator, will show that the average length of a session is really

$$(2 * 10 + 8 * 60)/10 = 50 \text{ min.}$$

As it turns out, the average length of a session is not the average of the individual student averages.

All programs should be divided into procedures, and the procedures, as well as the interaction between procedures, should be tested separately. This general technique was described in more detail in the section of Chapter 5 entitled "Testing Procedures." Below, we give some hints on how to test and debug the individual procedures.

*syntax  
errors*

Programming errors can be divided into three classes: syntax errors, run-time errors, and logical errors. A *syntax error* occurs when a portion of the program violates the syntax rules of the programming language, for example, when it is missing a required semicolon or an *end*. These are usually easy to find. The compiler will discover them and produce an error message. The error message may not accurately describe the nature or location of the error, but it definitely does indicate an error, and the location given is very likely to be the approximate location of the error.

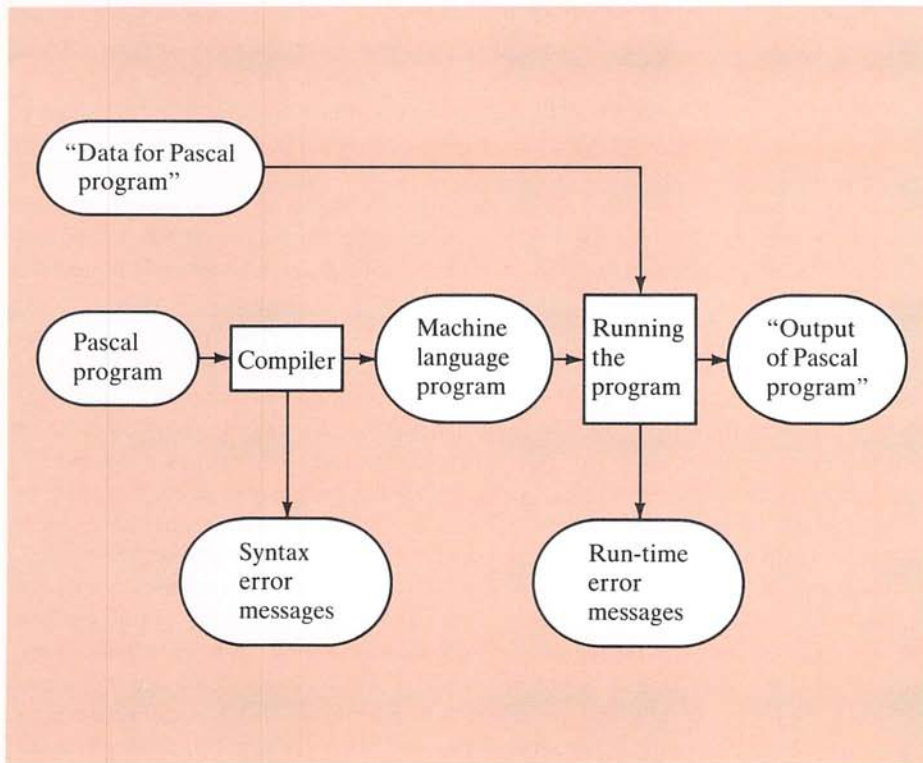
*run-time  
errors*

*Run-time errors* also result from the program violating the rules of the language, in our case Pascal. However, they are not syntax errors and so are not discovered by the compiler. As illustrated in Figure 12.2, the two types of errors are discovered at different stages of program processing. A run-time error is detected only when the program is run. For example, the statement below will produce a run-time error if the value of N is 0:

$X := 1/N$

To find out whether or not it is an error, it is necessary to run the program to see if N does receive the value 0. Run-time errors also produce an error message that gives some hint of the location and nature of the error. They are harder to find than syntax errors but still have the advantage that they are likely to produce an error message, although they are not guaranteed to do so. This is because they may only occur for certain inputs and not for others. For example, the above line of code will not produce a run-time error on those inputs that cause N to take on any value other than 0. (Addi-





**Figure 12.2**  
Syntax and run-time errors.

tional discussion of these two types of errors can be found in the section of Chapter 3 entitled “Testing and Debugging.”)

The third type of error is the hardest to locate. *Logical errors* are errors in the algorithm or in the translation of the algorithm from pseudocode to Pascal. They are difficult to find because the computer does not give any error message. They occur when the program is a perfectly valid Pascal program that performs a perfectly valid computation and gives out an answer. The only problem is that the answer is sometimes wrong. For example, the following piece of code is supposed to output the average of the elements in the array *A* of type *array[0 . . 10] of integer*:

```

Sum := 0;
for I := 0 to 10 do
    Sum := Sum + A[I];
writeln(Sum/10)

```

This is a perfectly valid piece of Pascal code and will run with no problems or error messages. For most values of the array elements, the answers will even look “about right.” However, there is a logical error. Since the array bounds go from 0 to 10, there are 11 elements, not 10. Hence, the last line should be

```

writeln(Sum/11)

```

*logical errors*

When a logical error is discovered, it is usually not too difficult to fix. The problem is finding such errors. Since the computer produces no error messages for logical errors, it is difficult to be certain that a program contains no errors of this nature.

Correcting run-time and logical errors is a three-stage process. The first stage is error avoidance. If a program is carefully designed along the lines we have suggested in this book, then the number of such errors should be few. The second stage is testing. In the testing stage, each procedure should be given a separate test. Testing procedures separately serves to tell which procedure or procedures contain the mistakes. The last stage is debugging. In that stage, the exact nature of the error is determined and the error is corrected.

*choosing  
test data*

When testing a procedure, you want to find some input values or parameter values that will expose possible errors. One way to catch an error is to find input values for which you know the correct output—perhaps by doing the calculation in some other way or by looking the answer up. Then you can run the procedure or program on those values. If the procedure's answer differs from the correct answer, then you know there is a mistake in the procedure. However, always remember that just because a procedure or program works correctly on 10 or even 100 test cases, this is not a guarantee that it does not contain an error. It might make a mistake on the next new input that is tried.

*boundary  
values*

One other technique to increase your confidence in a program is to use a variety of different types of test data. For example, if the data is an integer, try a large positive number, a small positive number, zero, a small negative number, a large negative number, and any other categories that you can think of. For loops, try data that will cause the loop to be executed zero, one, and more than one times (or as many of those cases as are possible for the loop in question). Be sure to use a representative sample of the possible *boundary values* as test data. There is no precise definition of the notion of a boundary value, but you should develop an intuitive feel for what it means. If a loop will be executed some number of times between 1 and 10, depending on the data, then be sure to have a test run that executes it 1 time and one that executes the loop 10 times. If a procedure does something to only one element of an array, always test the first and last elements of the array. Of course, you should also test a “typical” (nonboundary) value.

*fully  
exercising  
code*

Still another testing technique consists of *fully exercising* the procedure or program. This technique consists of using a collection of test cases that will cause each part of the procedure to be executed. This means executing each statement and sub-statement, and also making each boolean expression that controls a loop assume the value true at least once and assume the value false at least once. For example, consider the following piece of code:

```
if X > 0 then
    Procedure1
else if Y <= 0 then
    Procedure2
else if Z >= 0 then
    Procedure3
else
    Procedure4
```

```
if <boolean exp> then
    <statement 1>
else
    <statement 2>
```

To test both statements requires at least two test runs, one that makes <boolean exp> true and one that makes it false.

```
while <boolean exp> do
    <statement>
```

There should be at least two test runs: one that makes <boolean exp> false and so skips the loop and one that makes it true and so executes at least one iteration of the loop.

**Figure 12.3**  
**Fully exercising**  
**code.**

The call to Procedure4 will never take place unless the test data causes X to be less than or equal to zero and at the same time causes Y to be greater than zero and at the same time causes Z to be less than zero. The call to Procedure3 will never take place unless the test data causes X to be less than or equal to zero and at the same time causes Y to be greater than zero but at the same time causes Z to be greater than or equal to zero. Fully exercising a program is difficult to do. However, if the program is divided into procedures and the procedures are small, then it is possible to fully exercise each procedure separately. Figure 12.3 illustrates some other examples of fully exercising code.

An even better test of a program can be obtained with a technique called *testing all paths*. When you are testing all paths, the tests must not only cause each statement to be executed at least once but must also cause each possible combination of branch and loop behaviors to take place. Some combinations may be impossible to achieve, but the test set should cause all other combinations to occur.

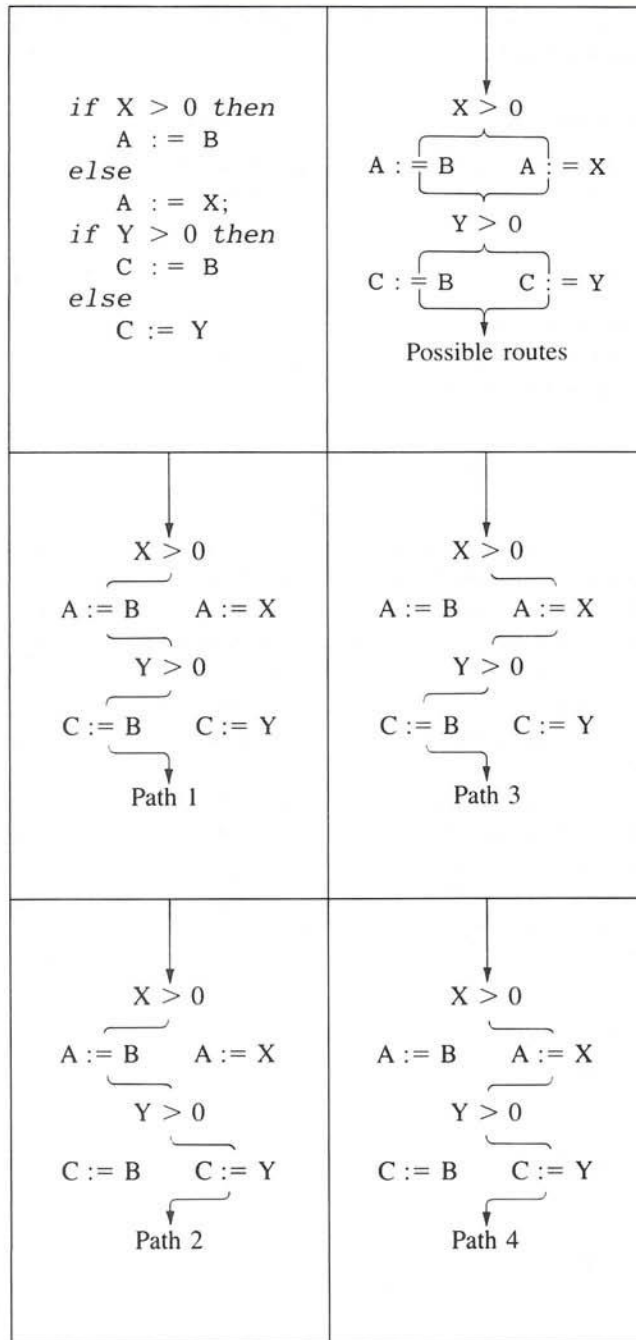
To see the difference between fully exercising a piece of code and testing all paths, consider the following code:

```
if X > 0 then
    A := B
else
    A := X;
if Y > 0 then
    C := B
else
    C := Y
```

This piece of code can be fully exercised with two inputs, one in which both X and Y are positive, and one in which they are both negative. To test all paths requires four tests, as shown in Figure 12.4. Testing all paths is a good testing strategy. Unfortunately, it is often difficult to find a set of test cases that will test all paths. Frequently you must settle for fully exercising the procedure and then testing as many paths as you reasonably can.

*testing  
all paths*





**Figure 12.4**  
Testing all paths.

Testing can tell you that a program or procedure contains an error. Moreover, if it is done correctly, it will tell you which procedure contains the error, but it usually will not tell you what the error is. Once you know that a procedure contains an error, you still must correct the error. This is the debugging stage. At this stage, frustrated programmers are sometimes tempted to make random changes, in the hope that these changes will magically correct the errors. They seldom do, and moreover, they make the program even less understandable. The way to correct an error is to test and analyze until you have located the exact cause of the error. Once the cause is located, the cure is usually easy to find.

*debugging*

One of the best ways to find the exact location of an error is to watch the procedure while it performs its calculation; that is, to watch the values of the various variables changing. This is called *tracing*. When a variable is traced, its value is written out either every time it is changed or at some other specified times. Some systems provide special debugging facilities for doing this automatically. If your system does not, then you can use temporary `writeln` statements in the manner described in the section of Chapter 7 entitled “Debugging Loops.”

*tracing*

---

## Verification

In the ideal world, there would be no need for testing and debugging, because all programs would be correct. In the ideal world, the programmer would prove the correctness of a program in much the same way that a mathematician proves a theorem. Proving the correctness of a program, that is, proving that it does what the specifications say it is supposed to do, is called program *verification*.

Whether or not it is practical to prove the correctness of large programs is still a hotly debated issue. In practice, testing and debugging will never be completely eliminated, but they may become less necessary, and program verification may become a more common practice than it is today. For small programs, verification is usually a realizable task. The debate arises when the discussion turns to large programs. There certainly is little hope of verifying a large program, such as a compiler, if you are given just the program code (without very extensive comments) and the specifications. On the other hand, if the top-down design strategy is used and each piece is verified separately, then it may be a tractable task. In this scenario, the verification takes place as the program is written, not after it is finished.

The debate over verification is one of degree. How formal should the verification be, and to what extent can it be relied on? Certainly a programmer should always make a serious attempt to somehow demonstrate (at least to himself or herself) that the algorithms are correct and that the code accurately represents the algorithms. Code should never be designed by simply writing down something that “looks like it might work” and then running a few test cases “to see if it works.”

We will not discuss verification in any great detail. If you would like to get a feel for what a proof of correctness looks like, read the optional section of Chapter 7 entitled “Invariant Assertions and Variant Expressions.”

## Portability

A *portable* program is one that can be moved from one computer system to another with little or no change. Since programs represent a large investment in programmer time, it pays to make them portable.

*standard  
language*

One way to make a program portable is to adhere closely to a *standard version* of the programming language. There are national and international organizations that set standards for the syntax and other details of programming languages. All the Pascal programs in this book conform to the ANSI/IEEE standard and to the ISO standard. (The initials have the following meanings: ANSI, American National Standards Institute; IEEE, Institute of Electrical and Electronic Engineers; ISO, International Standards Organization.)

As impressive as all those initials may look, it is still not true that all the Pascal programs in this text will run on any system that claims to follow the Pascal standards. There are two reasons for this: The standards leave some details unspecified, such as exactly how many characters the system will have and the value of `maxint`; also, real systems usually differ at least slightly from the prescribed standard. Despite these shortcomings, standards do serve a purpose. If you adhere to the versions of Pascal described by them, then your program will be more portable. For example, if your system initializes all `integer` variables to zero, do not use this feature. Instead, always explicitly initialize all variables. That way your programs will also work on systems that do not initialize variables. (None of the standards require that variables be initialized and, in fact, most Pascal systems do not do so.)

Sometimes implementation-dependent details are unavoidable. For example, `TURBO Pascal` does not always follow the official standards laid down by the standards organizations. Throughout this book we have indicated a number of ways in which `TURBO Pascal` differs from, or extends, standard Pascal. In order to make your program more portable, isolate all such implementation-dependent details into clearly labeled procedures. Then, when the program is moved, the things that need to change are easy to find and easy to modify. Input and output tend to be very implementation-dependent. Hence, input and output should always be isolated into procedures.

---

## Efficiency

*time  
and  
storage*

The *efficiency* of a program is a measure of the amount of resources consumed by the program. Traditionally, the only resources considered have been time and/or storage. The less time it uses, the more time-efficient a program is. The less storage it uses, the more storage-efficient it is.

When you are running small programs of the type you encounter when first learning to program, efficiency is not usually an issue. If the user waits a few extra seconds for an answer, that is insignificant. A small program is also unlikely to use more than a very small fraction of the available storage. Moreover, if the program is run just a few times, then the savings are likely to be minimal at best. However, when you are running large programs repeatedly over a long period of time, the amount of time and storage

---



saved can be significant. Computer time and storage cost money. Hence, if a program can be changed so that it runs faster or uses less storage, then, *all other things being equal*, the change should be made.

In some specialized settings, efficiency is critically important. If the computer is controlling a hospital patient-monitoring system, a fraction of a second delay may mean the patient's life. A very sophisticated storm-predicting program is useless if it takes two hours to predict that a tornado will arrive in one hour. A program to work in a small wrist calculator or a small satellite may have to make do with very little storage.

To illustrate the notion of efficiency, consider the task of searching a nonempty array of integers to see whether or not a particular integer is in the array. If the array is called *A*, and the integer being searched for is called *Key*, then the following loop will accomplish the task (*First* . . . *Last* is the array index type):

*example—  
searching  
an array*

```
I := First;
while (Key <> A[I]) and (I < Last) do
  I := I + 1;
if Key = A[I] then
  writeln(Key, ' is in the array.')
else
  writeln(Key, ' is not in the array.')
```

The loop checks each element of the array until it either finds the value *Key* or gets to the end of the array without finding *Key*.

Now let us suppose that we know the array elements are ordered so that

$$A[\text{First}] < A[\text{First} + 1] < A[\text{First} + 2] < \dots < A[\text{Last}]$$

In this case, we can make the loop run more efficiently, in the sense of taking less time. If we know the list is ordered, we can stop looking for the value *Key* as soon as the following holds:

$$\text{Key} \leq A[I]$$

This is because if *Key* equals *A[I]*, we have found the value of *Key*, and if *Key* is less than *A[I]*, then we know that *Key* is smaller than all the array elements that follow and so cannot possibly equal any of them. Hence, assuming that the list is ordered, we can replace the boolean expression in the *while* loop with the following:

$$(\text{Key} > A[I]) \text{ and } (I < \text{Last})$$

With this second boolean expression, the loop will frequently perform fewer iterations. If the value of *Key* is in the array, then the two loops perform exactly the same number of iterations, but if *Key* is not in the array, then, “on the average,” the second loop will perform a little more than half the number of iterations that the first one does. A precise definition of what we mean by “on the average” is beyond the scope of this book. However, the important thing to observe is that the second version does save a large fraction of time on a large number of inputs. The program in Figure 12.5 illustrates the savings for one particular set of values.

Occasionally, a simple change like the one we just described can improve efficiency significantly. More often, however, the savings due to a minor change are corre-

**Program**

```

program Compare(input, output);
{Compares two search algorithms.}
const First = 1;
      Last = 10;
type Index = First .. Last;
      List = array[Index] of integer;
var A: List;
      I: Index;
      Key: integer;

procedure Search1(A: List; Key: integer);
{Outputs a message saying whether or not Key is
the value of an array indexed variable A[I], for some I.}
var I: Index;
begin{Search1}
  I := First;
  while (Key <> A[I]) and (I < Last) do
    I := I + 1;
  if Key = A[I] then
    writeln(Key, ' is in the array.')
  else
    writeln(Key, ' is not in the array.');
```

writeln('Search 1 executed ', I - First, ' loop iterations.')

```

end; {Search1}

procedure Search2(A: List; Key: integer);
{Outputs a message saying whether or not Key is
the value of an array indexed variable A[I], for some I.
Precondition: A[First] < A[First + 1] < ... < A[Last].}
var I: Index;
begin{Search2}
  I := First;
  while (Key > A[I]) and (I < Last) do
    I := I + 1;
  if Key = A[I] then
    writeln(Key, ' is in the array.')
  else
    writeln(Key, ' is not in the array.');
```

writeln('Search 2 executed ', I - First, ' loop iterations.')

```

end; {Search2}

```

**Figure 12.5**  
**Comparing two**  
**algorithms.**

```
begin{Program}
  writeln('Enter 10 integers in ascending order: ');
  for I := First to Last do
    read(A[I]);
  readln;
  writeln('Enter a Key to search for: ');
  readln(Key);
  Search1(A, Key);
  Search2(A, Key)
end. {Program}
```

### Sample Dialogue

```
Enter 10 integers in ascending order:
2 4 6 8 10 12 14 16 18 2001
Enter a Key to search for:
11
    11 is not in the array.
Search 1 executed      9 loop iterations.
    11 is not in the array.
Search 2 executed      5 loop iterations.
```

**Figure 12.5**  
(continued)

spondingly minor. To make substantial savings, a completely new and more complicated algorithm is usually required. In the next section we discuss the advisability of using a complicated, efficient algorithm as opposed to a simple, less efficient one.

---

## Efficiency versus Clarity

The current trend is to pay less attention to time and storage efficiency. The reasons for this switch are that computer time and storage have become less expensive, while programmer time has become more costly. It simply does not make sense to pay thousands of dollars (or even much less) in programmer salaries in order to realize a few dollars of savings in computer usage.

Frequently there are also other hidden costs in making a program very "efficient." A typical way to arrive at an "efficient" program is to start with a simple, easy-to-understand, correctly running program and then to make changes to the program so that it runs faster or uses less storage. In the process of doing so, a number of unfortunate things can happen. The changes may introduce an error. That produces the ultimately inefficient program. Getting the wrong answer quickly is never a bargain. Changing a program to make it run faster may make the program harder to understand. At some later time, when the program needs to be changed, this will increase the time needed to change it and will make errors more likely. Large programs typically have a life span in which they are modified numerous times by a series of different programmers. In that situation, clarity is the critically important consideration. When the choice is between clarity and efficiency, it usually pays to choose clarity.

---



A good strategy is the following. First, make sure the program is clear and correct. Within those constraints and the constraints of available programmer time, it pays to make the program more time- and storage-efficient.

---

## Off-Line Data

Thus far we have emphasized interactive computing environments consisting of a video screen and keyboard. In that mode the program from time to time requests data of the user and from time to time produces output, typically intermixing input and output. Computing is sometimes done in what is called *batch mode*. In batch mode, the program and all the data are fed into the computer at one time, and the output is not available until after the program has run to completion.

*files*

Data can be stored in memory in units called *files*. Programs can then read all or some of their data from these files. Data kept in files is called *off-line* data. In fact, data in any form that can be read by the computer without being entered by the user while the program is running is called *off-line* data. Data kept on punched cards, magnetic tape, and various sorts of disks are all examples of off-line data. In batch processing, all of a program's data is off-line data. In other situations, some of the data is off-line data, and the rest is entered from the keyboard by the user.

As you might guess, programs written to read off-line data differ from the kinds of interactive programs we have seen primarily in how they handle input and output. With off-line data, the program need not prompt the user to enter data; the data is always there waiting to be read. A second difference is that, with off-line data, the program can count on the data always being in a fixed format. It need not write out detailed input instructions or echo input or have elaborate routines to recover from errors in entering the input. On the negative side, in order to prepare off-line data, the user must have at least some technical knowledge about formatting and entering the data.

The modern trend is toward a combination of off-line and interactive use. Some tasks, such as an airline reservation system, naturally demand that the program interact with the user. Other tasks, such as processing lists of numbers, perform more naturally if the data collection is prepared ahead of time and the program reads this data from a file. A combination of off-line and interactive use can often provide the best of both worlds. For many tasks, the bulk of the data can be prepared beforehand and stored in a file; the program can then interact with the user via a keyboard and screen to let the user know what is happening and to receive directions on how to process the data. If there are large amounts of output, the program directs it to a file or to some output device, such as a printer. That way the output can be studied a leisure. Chapters 13 and 16 deal with programs of this mixed variety.

---

## Coping with Large Programming Tasks

*what is  
"large"?*

The programs presented in an introductory programming book are extremely small compared to the programs typically worked on by professional programmers. One widely used Pascal compiler consists of almost 7400 lines of high-level-language code. If written as small as the text of this book, that program would occupy over 130 pages!

---

Programs that large differ qualitatively as well as quantitatively from the sorts of small programs we have seen. Like the servant in the Dickens story (quoted at the end of this section), the typical reader of this book has only had a “sip” of programming. It is a larger sip than the servant had, and the reader who completes this book will have tasted programming. However, large programs really do have a different flavor from small programs. All the design principles we have discussed, such as modularity and top-down design, are even more important when you are designing large programs. But there are some additional considerations that only come into play when programs are very large.

A program as large as a compiler or a complete operating system is not written by a single person. The effort is too large for any single individual. To take an extreme case, F.P. Brooks reports that the design of the IBM OS/360 operating system consumed 5000 “man-years.” That figure includes support staff and probably would be lower today. However, it is clear that the job is too large for any single programmer. The production of a piece of software of that size requires a major organizational effort. It is a management feat as well as a design feat. The book by Brooks, cited at the end of this chapter, gives a good description of the management problems involved in designing large programs. Most of the book is understandable to anyone who has read the first few chapters of this text.

Another feature of large programs is that they represent a large investment of both time and money. Hence, if they are written clearly enough to be easily modified and adapted, rather than discarded in favor of a completely new program, then significant time and money can be saved. In fact, most large programs, like compilers and operating systems, are continually being changed. Bugs are discovered and must be fixed. New features are added. However, even large programs eventually become out of date or deteriorate in quality. Unless a large program is well written and suitable for the task at hand, then, just as in the case of a small program, it is more efficient to write a new program than it is to fix the old one.

*group  
effort*

---

“Did you ever taste beer?”  
“I had a sip of it once,”  
said the small servant.  
“Here’s a state of things!”  
cried Mr. Swiveller. . . .  
“She *never* tasted it—  
it can’t be tasted in a sip!”  
*Charles Dickens, The Old Curiosity Shop*

---

---

## Summary of Terms

### batch processing

As used in this book, running a program in batch mode means entering the program and all its data at one time and waiting until the program finishes before receiving the output.

---

**data abstraction**

Disregarding (“forgetting”) the features of a data structure that are irrelevant to the problems, algorithms, or programs under consideration.

**efficiency**

The efficiency of a program is measured by the amount of resources that the program consumes. The less resources it consumes, the more efficient it is. Time and storage are the resources that are usually considered.

**fully exercise**

A technique for testing a piece of code (such as a program or procedure). It consists of finding a set of test inputs such that running the program on the test inputs will cause each statement and substatement to be executed on at least one of the test runs and will cause each boolean expression that controls a loop to assume the value `true` on at least one run and `false` on at least one run.

**logical error**

A program error that is due to an error in the algorithm or an error in translating the algorithm into the programming language. Normally, logical errors produce no error messages.

**off-line data**

Data in any form that can be read by the computer without being entered by the user while the program is running is called off-line data. (For example, data kept in files is off-line data.)

**portability**

A program is portable if it can be moved from one system to another with little or no change.

**procedural abstraction**

Disregarding (“forgetting”) the features of a procedure that are irrelevant to the problems, algorithms, or programs under consideration.

**run-time error**

A program error that is discovered by the computer system at the time the program is run. See *syntax error*.

**standard version of a programming language**

A version of a programming language that is defined by some official standards organization.

**syntax error**

An error consisting of a violation of the syntax rules of a language. Syntax errors are discovered and reported by the compiler. See *run-time error*.

---



### testing all paths

A technique for testing a piece of code (such as a program or procedure). It consists of finding a set of test inputs such that running the program on the test inputs will cause each possible combination of branch and loop behaviors to occur on at least one of the runs.

### tracing

Inserting `write` or `writeln` statements into a program so that the values of the variables will be written out as the program performs its calculations. Some systems have debugging facilities that do this automatically.

### verification

Verifying a program means proving that it meets the specifications for the task it is supposed to perform.

---

## Exercises

### Self-Test Exercises

1. The following piece of code is supposed to set `Sum` equal to the sum of the first 100 positive numbers. It contains a bug. What is it?

```
Sum := 0;
I := 1;
repeat
  Sum := Sum + I;
  I := I + 1
until I >= 100
```

If you cannot find the bug, replace 100 with 3 and trace the computation, either with pencil and paper or by embedding the code and a `writeln` statement in a program.

2. Choose some input values to fully exercise the following piece of code (all the variables are of type `integer`):

```
readln(X, B, C)
if X >= 5 then
  A := B
else
  A := C;
Sum := 0;
while X > 0 do
  begin
    Sum := Sum + X;
    X := X - 1
  end;
```

3. Choose some input values to test all paths in the code of the previous exercise.
4. What is wrong with the following program? (It correctly computes the average of 10 integers.)

```
program Exercise(input, output);
type Index = 1 . . 10;
    List = array[Index] of integer;
var A: List; I: Index; Sum: integer; Average: real;
begin{Program}
    writeln('Enter 10 integers: ');
    for I := 1 to 10 do
        read(A[I]);
    Sum := 0;
    for I := 1 to 10 do
        Sum := Sum + A[I];
    Average := Sum/10;
    writeln('The average is: ', Average)
end. {Program}
```

### Interactive Exercises

5. Redesign the algorithm for computing average terminal session times that was given in the pseudocode in the section entitled “Testing and Debugging.” As noted in the text, the pseudocode contains a logical error.
6. If you have access to more than one computer with Pascal available, run some of your programs from previous exercises on two or more machines. Is it necessary to change the programs in any way?

### Programming Exercises

7. Write a program to read in a list of 100 or fewer integers and then do any of the following, as the user requests: display the list in sorted order from largest to smallest; display the list in sorted order from smallest to largest; compute the average; compute the mean; compute percentiles (e.g., the tenth percentile is the 10% of the scores that are highest in value); list the scores in the order largest to smallest or smallest to largest, showing how much each differs from the average and/or median. Do this jointly with two or three other people. Each person should do separate subtasks, and the code should be integrated into a single program.
  8. (To do this one, you and another group must have done the previous exercise.) The two groups exchange programs and then each modifies the other's program as follows: The option of computing the standard deviation is added to the program (see Exercise 21 in Chapter 4 for definitions). The user is also given the option of listing the scores in order, either lowest to highest or highest to lowest, with an annotation indicating whether or not the score is within one standard deviation of the average.
-

## 9. A polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

can be evaluated in a straightforward way by performing the indicated operations and using the procedure `Power` given in Figure 8.3. An alternative method is to factor the polynomial according to the following formula, known as Horner's rule:

$$(\cdots ((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$$

Write two procedures to evaluate polynomials by these two different methods. The procedures will read  $n$  as well as the coefficients  $a_n$ ,  $a_{n-1}$ , and so forth from the keyboard. After completely debugging the procedures, insert extra code to count the number of addition and multiplication operations performed. (Do not forget to count addition and multiplication operations performed by all the procedures such as `Power`.)

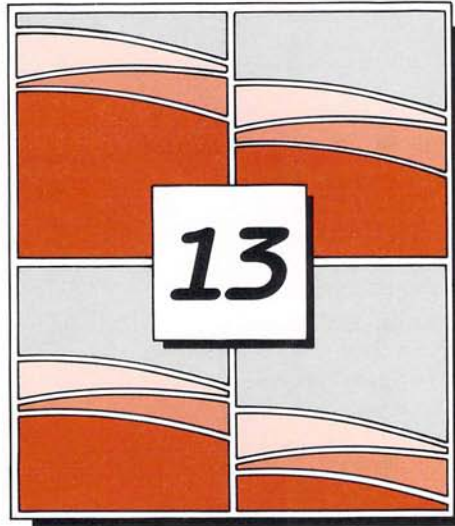
---

## References for Further Reading

- American National Standard Pascal Computer Programming Language*, 1983, Institute of Electrical and Electronic Engineers, New York. A very technical document that is not easy to read, but you may find it interesting to look through.
- J.L. Bently, *Writing Efficient Programs*, 1982, Prentice-Hall, Englewood Cliffs, N.J. A good source for more information on writing efficient programs.
- F.P. Brooks, *The Mythical Man-Month*, 1975, Addison-Wesley, Reading, Mass. A good collection of essays to give you a feel for the problems involved in writing very large programs.
- O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, 1972, Academic Press, New York. Good essays on programming techniques, data structures, and programming style.
- E.W. Dijkstra, *A Discipline of Programming*, 1976, Prentice-Hall, Englewood Cliffs, N.J. A good series of essays on programming techniques and programming style.
- D. Gries, *The Science of Programming*, 1981, Springer-Verlag, New York. A good source for more information on program verification.
- John Guttag and Barbara Liskov, *Abstraction and Specification in Program Development*, 1986, MIT Press, Cambridge, Mass; McGraw-Hill, New York.
- Gerald M. Weinberg, *The Psychology of Computer Programming*, 1971, Van Nostrand Reinhold, New York. A collection of essays on the subject described by the title.
- Edward Yourdon, *Classics in Software Engineering*, 1979, Yourdon Press, New York. A collection of famous essays on programming methodology.
-







## ***Text Files and Secondary Storage***

Polonius: What do you read, my lord?

Hamlet: Words, words, words.

*William Shakespeare, Hamlet*

## Chapter Contents

### Part I

#### Text Files

#### Writing and Reading Text Files

#### TURBO Pascal—Opening Files

#### TURBO Pascal—More about File Names (Optional)

#### Standard Pascal—File Handling

#### Pitfall—Mixing Reading and Writing to a Text File

#### Pitfall—The Silent Program Self-Test Exercise

### Part II

#### read and write Reexamined

#### eof and eoln

#### Using a Buffer

#### Pitfall—Forgetting the File Variable in eof or eoln

#### Pitfall—Use of eoln and eof with Numeric Data

#### Text Files as Parameters to Procedures

#### Pitfall—Portability

#### Basic Technique for Editing Text Files

#### TURBO Pascal—Temporary Files

#### TURBO Pascal Case Study—Editing Out Excess Blanks

#### Text Editing as a Programming Aid

#### Summary of Problem Solving and Programming Techniques

#### Summary of Pascal Constructs Exercises

**T**his chapter describes how Pascal programs can store information in units called *text files* so that other Pascal programs may read the information at a later time. File handling is highly implementation dependent. The basic concepts of file handling are the same for all Pascal systems, but the details will vary from one system to another. In particular, TURBO Pascal and standard Pascal differ in how they handle text files. This chapter emphasizes text file manipulation by TURBO Pascal programs. We include a section which describes those standard Pascal details that differ from those of TURBO Pascal, but with one short exception, all of the programming examples are TURBO Pascal programs. These programs will run without change on TURBO Pascal systems but would require some minor modifications before they would run on a standard Pascal system. The chapter sections that are labeled neither “Standard Pascal” nor “TURBO Pascal” apply to both systems.



# PART I

Chapter 1 presented a description of the main components of a computer. That description emphasized what is called *main memory*. Virtually all computers have an additional form of memory called *secondary storage* or *external storage*. On a personal computer, this secondary storage is likely to be a device called a *floppy disk drive*. On a large system, it is likely to be a device called a *hard disk drive*. As the names imply, these storage media are disk shaped. They are similar to phonograph records in that they store information on tracks of a disk and read the information via an arm that rests over the disk. However, their physical properties are closer to those of the magnetic tape commonly used to record music than they are to those of a phonograph record. In any event, it is their characteristics as viewed by the programmer that are important to us. We will not need to know about the physics of how they work.

*secondary  
storage*

*disks*

Main memory is often of the type called *volatile*, which means that when you shut off the computer, the data stored in memory goes away. In fact, for all practical purposes, the data goes away as soon as the program ends. Secondary storage is *non-volatile*. It can be used to store data for as long as is needed. In this chapter we will describe a method whereby a Pascal program can store text data in secondary storage. We will also describe how another program can access that data.

---

## Text Files

In Pascal, a *file* is a named collection of data in secondary storage. The important properties of a file are that a program can write data to it, that it can remain in storage after the program has finished running, that it has a name, and that other programs can read it later on. In a later chapter we will describe all the types of files that are available in Pascal. For now we will only discuss one special kind of file called a "text file."

*file*

A *text file* is a file that contains the same sort of data as the output screen. More precisely, a text file contains a stream of characters divided into units called *lines*. One way to think of a text file is as if it were a very long sheet of paper that a program can write on and that the same or a different program can later read. These conceptual sheets of paper are divided into lines in the same way that output to the screen is divided into lines. However, unlike screens and sheets of paper, there are no limits to the size of a text file. There is no limit to the number of characters on a line in a text file, and there is no limit to the number of lines (as long as the file does not use all of the computer's available secondary storage).

*text  
file*

Inside a Pascal program, a text file is referred to by means of a Pascal identifier that is declared as if it were a variable of a type called *text*. For example, the following declares *Stuff* to be of type *text* and would occur in the variable declaration section of a program:

*the type  
text*

```
var Stuff: text;
```

---

*file  
variables*

Identifiers such as `Stuff` are called *file variables*. Although a file variable is a kind of variable, it is a very atypical kind of variable. It cannot appear in an assignment statement, nor can it be used in many of the other ways that the usual kinds of variables are used. It has special standard procedures and special syntax rules of its own. It is probably better simply to think of it as a name for a file and not think of it as a variable at all.

*other names  
for files*

Text files are used for many purposes that do not involve the Pascal language. Any file that you create or read with the usual editor is a text file. Hence, a text file can exist and have a name before it is used by a Pascal program. On some systems, this name might not even satisfy the Pascal syntax for identifiers. To accommodate such file names, many versions of Pascal, including TURBO Pascal, provide a mechanism to associate a file variable name with a file and thus rename it for the duration of the Pascal program. On those systems a file will have one name outside of the Pascal program, but will be referred to by a possibly different file variable name within the Pascal program. No matter how many names a text file may have, it is always referred to by its file variable name when reading or writing is performed by a Pascal program.

---

## Writing and Reading Text Files

*write  
and  
writeln*

Data is written to text files in the same way that it is written to the screen—that is, by `write` and `writeln` statements. In order to write to a text file, the `write` statement must contain the file variable associated with the file; otherwise, the output will go to the screen. For example, to write the string 'Hello' to the file `Stuff`, the following statement will suffice:

```
writeln(Stuff, 'Hello')
```

*specifying  
the file*

Unfortunately, this syntax is confusing. The identifier `Stuff` looks like a variable whose value is to be written out. It is not. It is the file variable name of the file to which the output is being sent. There is no way to determine this by looking only at the `writeln` statement. The only way that you, or the compiler, can figure this out is to look at the variable declaration section. If `Stuff` is declared to be of type `text`, then this first argument names the file that is to receive the output. If, on the other hand, we had declared `Stuff` to be a variable of some type such as `char` or `integer`, then this statement would instead write the value of `Stuff`, as well as the word 'Hello', to the screen.

When reading or writing files, a Pascal program handles numbers in the same way it handles numbers entered from the keyboard or written to the screen. The system automatically converts numbers to characters when writing numbers and automatically converts characters to numbers when reading into variables of type `integer` or `real`. A text file can contain nothing but characters divided into lines. It cannot contain numbers, but because of automatic type conversion, the following statement will cause no problems:

```
write(Stuff, 5)
```

---



Just as with output to the screen, numbers are handled as you would hope. The system changes the number 5 to the character '5', and it is the character '5' that is written into the text file.

Text files can be read by means of `read` and `readln` statements. There are problems associated with mixing `read` and `write` statements to the same text file. Therefore, for now we will assume that different programs are doing the reading and the writing. Suppose another program contains the following declarations:

```
var Stuff: text;  
    FirstLetter, SecondLetter: char;
```

This other program can then prepare the same file `Stuff` for reading. (We will explain how later on.) After that, the program can read from the file by executing a `read` statement such as the following:

```
read(Stuff, FirstLetter, SecondLetter)
```

Since the first thing in the file is the word 'Hello', this will set the value of `FirstLetter` to 'H' and the value of `SecondLetter` to 'e'. Reading starts with the first character of the first line of the file and proceeds through the file. The first `read` statement reads the first so many characters, the next `read` executed reads the next so many characters, and so forth. There is no way to backspace.

The statements `read` and `readln` behave the same for text files as they do for the sort of keyboard-entered data we have used so far. The only difference is that if a text file variable is given as the first argument, then the data is read from the indicated text file rather than from the keyboard. Just as with a `write` statement, this is confusing syntax. The only way to tell that the first argument of a `read` or `readln` refers to a text file is to look at the declaration section to see if it is declared to be of type `text`.

*read  
and  
readln*

## TURBO Pascal

### Opening Files

In TURBO Pascal a text file has two kinds of names. Each file has a single name which all programs that refer to the file use to identify it. This name is a string called its *string name*. (String names are also called *directory names*.) The string name of a file is the "ordinary" name for the file. It is the name you have been using up until now. This string name is the name used for the file within the TURBO environment. If you list the files in your directory with the `d` command, the file names you see are string names. Typically this string name ends in '.TXT', indicating that the file contains *text*. For example, 'DATA.TXT' might be a string name for a file containing data of some sort.

*string name*



Other three-letter suffixes will also work. In particular, '.PAS' will work with no problems. In addition to the string name, each TURBO Pascal program that uses a text file gives the file a second name that is used to refer to the file within that program. This other name is a file variable that has been declared to be of type `text`. In order to read a text file or to write to a text file, a TURBO Pascal program must first associate a file variable name with the string name of the file. This is done with the predefined procedure `assign`. The syntax is as follows:

```
assign(<file variable>, <string value>)
```

*assign*

The first argument is a file variable that has been declared to be of type `text`. The second is a string value that is the string name of text file. For example, the following associates the file variable `TestFile` with the file whose string name is 'DATA.TXT':

```
assign(TestFile, 'DATA.TXT')
```

A call to `assign` is always the first thing that a program does with a text file. After the call to `assign`, the file is always referred to by its file variable name.

*opening  
files*

All files must be *opened* before a program can read from the file or write to the file. Opening a file instructs the system to prepare the file for reading or writing. In TURBO Pascal a text file is opened for either reading or writing, but not for both.

*rewrite*

A text file is opened for writing with the standard procedure `rewrite`. For example, the following opens the file named by the file variable `TestFile` for the purpose of writing to the file:

```
rewrite(TestFile)
```

Suppose that the above call of `assign` had been executed first. Then this file will be called 'DATA.TXT' in the directory and `TestFile` in the TURBO Pascal program. After executing this `rewrite` statement, the program may use `write` or `writeln` statements with the file. When doing so, the file is always referred to by the name `TestFile`. The `rewrite` procedure always gives a blank file, a "clean sheet of paper," so to speak. If there already is a file named 'DATA.TXT', then that file is erased. If there is no file called 'DATA.TXT', then the `rewrite` procedure will create one.

A complete example is given in Figure 13.1. That program will write the numbers 3 and 4 to a text file called 'DATA.TXT'. The numbers will be on two lines. If 'DATA.TXT' did not exist before the program was run, the program would create it. If there were a file called 'DATA.TXT', the previous contents of that file would be lost. After the program is run, the file will only contain the two numbers.

Figure 13.1 illustrates the use of the `writeln` statement both with and without a file variable as its first argument. The output from those `writeln` statements that have the file variable `TestFile` as their first argument will go to the file. On the other hand, the first and last `writeln` statements do not contain the file variable `TestFile`, and so their output goes to the screen. If those two `writeln` statements are omitted, the programs will output nothing at all to the screen.

---

**Program**

```

program Writer;
{Writes the numbers 3 and 4 to the text file 'DATA.TXT'}
var TestFile: text;
    N: integer;
begin{Program}
    writeln('Start program');
    assign(TestFile, 'DATA.TXT');
    rewrite(TestFile);
    writeln(TestFile, 3);
    N := 4;
    writeln(TestFile, N);
    close(TestFile);
    writeln('End of program')
end. {Program}

```

**Output to Screen**

```

Start program
End of program

```

**Output to 'DATA.TXT'**

```

3
4

```

**Figure 13.1**  
**TURBO Pascal**  
**program to write**  
**to a text file.**

In TURBO Pascal all files must be *closed* after the program has finished writing to or reading from the file. Closing a file tells the system that the program is through reading from or writing to the file. This is done with the predefined TURBO Pascal procedure `close`. As illustrated in Figure 13.1, `close` has one argument, which is the file variable name.

Be sure to note how the two types of file names are used in the program of Figure 13.1. The string name for the file is used only in the call to `assign`. Thereafter the file is always referred to by the file variable `TestFile`. Also note that the string name `'DATA.TXT'` is in quotes. That is because it is a constant of a *string* type. Any expression that evaluates to a *string* value may be used as this second argument. In particular, it is possible to use a variable of a *string* type. This use allows the program to read in a file name from the keyboard, as illustrated by the program in Figure 13.2.

A text file is opened for reading with the standard procedure `reset`. To set the stage for an example using `reset`, let us suppose that a TURBO Pascal program contains the following:

```
assign(TestFile, 'DATA.TXT')
```

*close*

*reading a file name  
from the keyboard*

*reset*

**Program**

```

program UsersChoice;
{Asks the user for a file name, creates a text file with that name,
and then copies one line of text from the keyboard to that text file.}
var FileVar: text;
    FileName: string[20];
    Line: string[80];
begin{Program}
    writeln('Enter a file name. If the file already');
    writeln('exists, its contents will be lost. ');
    readln(FileName);
    assign(FileVar, FileName);

    rewrite(FileVar);
    writeln('Enter a line of text: ');
    readln(Line);
    writeln(FileVar, Line);

    close(FileVar);
    writeln('That line is saved in the file ', FileName)
end. {Program}

```

**Screen/Keyboard Dialogue**

```

Enter a file name. If the file already
exists, its contents will be lost.
TEST.TXT
Enter a line of text:
A wet bird never flies at night!
That line is saved in the file TEST.TXT

```

**Output to 'TEST.TXT'**

```

A wet bird never flies at night!

```

**Figure 13.2**  
**TURBO Pascal**  
 program that reads  
 a file name from  
 the keyboard.

Then, the file which is called 'DATA.TXT' in the directory will be known as TestFile in the program. The following call to reset opens this file for reading:

```
reset(TestFile)
```

After executing the reset statement, the program may use read or readln statements with the file. When such statements are used, the file is always referred to by the file variable name, TestFile. A complete example is given in Figure 13.3.



**Program**

```

program Reader;
{Reads two numbers from the text file 'DATA.TXT', places them in variables
N1 and N2, and then outputs the contents of N1 and N2 to the screen.}
var TestFile: text;
    N1, N2: integer;
begin{Program}
    assign(TestFile, 'DATA.TXT');
    reset(TestFile);
    readln(TestFile, N1);
    readln(TestFile, N2);

    writeln(N1);
    writeln(N2);

    close(TestFile);
    writeln('End of program.')
end. {Program}

```

**Output**

(Assuming that the program in Figure 13.1 was run first.)

```

3
4
End of program..

```

**Figure 13.3**  
TURBO Pascal  
program to read  
from a text file.

## TURBO Pascal

### More about File Names (Optional)

In this section we assume that you are running TURBO Pascal on a DOS system. However, many of the remarks will apply, essentially unchanged, to other systems.

If you use the techniques given in the previous section without any changes, then all of the files manipulated by your TURBO Pascal programs must be on the default disk drive. (The default drive is usually called the “logged drive” and is the drive that is used by all programs and utilities, including the editor, whenever no drive name is specified.) It is also possible to specify a disk drive when giving a string name. This specification is made by inserting the name of the drive and a colon in front of the string name of the file. For example, the following associates the file variable name `TestFile` with the file on disk drive B whose string name is `'DATA.TXT'`:

```
assign(TestFile, 'B:DATA.TXT')
```

*naming the  
disk drive*

There should be no spaces around the colon. If you want to specify a file on drive A, or C, or some other drive, then use A or C or other appropriate disk drive name in place of B.

#### *path names*

DOS operating systems allow hierarchical directories of files consisting of files within subdirectories, as described in Appendix 5. These files within subdirectories are named by path names which give directory names as well as file names. A path name can be used as the string name for a text file within a TURBO Pascal program. For example, the following associates the file variable LetterVar with the file whose full path name is /DLETTERS/MOM.TXT:

```
assign(LetterVar, '/DLETTERS/MOM.TXT')
```

In the above example, the file assigned to the file variable is the file named MOM.TXT which is in the subdirectory of the root directory that is called DLETTERS.

The path name in the previous example applies to the default disk drive (i.e., the logged disk drive). If you wish to specify a disk drive, you may do so. For example, the following specifies that the file is on disk drive A:

```
assign(LetterVar, 'A:/DLETTERS/MOM.TXT')
```

---

## Standard Pascal—File Handling

In this section we will describe those aspects of standard Pascal text file handling that differ from what we have described for TURBO Pascal. In standard Pascal a text file often has only one name. In this section we will assume that the file variable name for a standard Pascal text file is the same as any other name that the file may have.

#### *program heading*

The file variable names for the files used by a standard Pascal program are listed in the program heading. We have already been doing that for the files called input and output. The keyboard and the screen are considered to be special kinds of files with the file variable names input and output. A program that reads from the text file ReadStuff and writes to the text file WriteStuff would start as follows:

```
program Sample(input, output, ReadStuff, WriteStuff);
```

The order of the file variable names is usually unimportant.

Even though the file variables are listed in the program heading, they must be declared in the variable declaration section. Hence, the following declaration must appear in the program:

```
var ReadStuff, WriteStuff: text;
```

The rest of the details for text file handling in standard Pascal are the same as those for TURBO Pascal, with two exceptions. First, standard Pascal includes no procedure like assign; in standard Pascal files normally have only one kind of name. Second, standard Pascal includes no procedure like close; files are closed automatically at the end of a program.

---

**Program**

```

program Writer(input, output, TestFile);
{Writes the numbers 3 and 4 into the text file TestFile.}
var TestFile: text;
    N: integer;
begin{Program}
    writeln('Start program');
    rewrite(TestFile);
    writeln(TestFile, 3);
    N := 4;
    writeln(TestFile, N);
    writeln('End of program')
end. {Program}

```

**Output to Screen**

```

Start program
End of program

```

**Output to TestFile**

```

3
4

```

**Figure 13.4**  
Standard Pascal  
program that  
writes to a  
text file.

The procedures `reset` and `rewrite` are used in the same way and have the same effect as what we described for TURBO Pascal. If a program has the heading and variable declarations described above, then the text file `WriteStuff` is opened for writing with the procedure call

```
rewrite(WriteStuff)
```

The text file `ReadStuff` is opened for reading with the procedure call

```
reset(ReadStuff)
```

As an example, Figure 13.4 contains a standard Pascal version of the TURBO Pascal program in Figure 13.1.

---

**Pitfall**
**Mixing Reading and Writing to a Text File**

A program cannot simply intermix reading and writing to the same file. At any one time, a text file that has been opened is available for either reading or writing but not both. To switch from writing to reading, the file must be reopened by a call to `reset`. To switch from reading to writing, the file must be reopened by a



call to `rewrite`. This is not a trivial restriction, since opening the file for writing completely erases the file. The typical sequence of actions is to open the file with `rewrite` and write to it, and then to have the same or a different program open the file any number of times with `reset` and read the file contents. The next time it is opened with `rewrite`, the file is erased.

One productive variation on this pattern of writing and reading is as follows: The file can be created or changed by a person using the editor to write to the file instead of having a Pascal program do the writing. Text files are the exact same kind of files as those used to hold Pascal programs, shopping lists, and term papers, and they can be edited with the editor just as a Pascal program can be edited.

## Pitfall

### The Silent Program

It is quite common to write a program so that all the data that is output is directed to a text file. If you do this, then the program will produce no output to the screen. This can be bewildering. If you write programs that way, the user may not even be able to tell when the program has finished. To let the user know what is going on, there should always be some output to the screen, even if it just says when the program starts and when it ends.

## Self-Test Exercise

1. Suppose the text file with the string name `STUFF.TXT` contains the following:

```
5 63 75
5 63 75
```

For example, you may have created the file using the editor, or it might have been created by a Pascal or other kind of program. What is the output of the following TURBO Pascal program?

```
program Exercisel;
const Space = ' ';
var Arthur: text;
    Number: integer;
    Letter1, Letter2: char;
begin{Program}
  assign(Arthur, 'STUFF.TXT');
  reset(Arthur);
  read(Arthur, Number); write(Number, Space);
  read(Arthur, Number); write(Number);
  readln(Arthur); writeln;
```

```
read(Arthur, Letter1); write(Letter1);  
read(Arthur, Letter1, Letter2);  
write(Letter1, Letter2);  
close(Arthur)  
end. {Program}
```

### Interactive Exercises

2. Write a program that writes your name to a text file. Using the editor, look at the text file after the program is run.
3. Create a text file using the editor (as you do when you write a program) and write three numbers in the text file. Write a program to read the three numbers and write them to the screen. After running the program once, go back and change the numbers. Then rerun the program.
4. Using the editor, change the text file from the previous example so that the three numbers are two digits long and have one space between them, and so that there are no spaces at the front of the line. For example, the text file might contain

25 36 47

Rerun the program from the previous exercise. Next, write a slightly different program that reads three characters from the same text file into three variables of type `char` and then outputs them to the screen. Run this new program on the same text file.

## PART II

---

The Moving Finger writes; and, having writ,  
 Moves on: nor all your Piety nor Wit  
 Shall lure it back to cancel half a line,  
 Nor all your Tears wash out a Word of it.

*Omar Khayyam, The Ruba'iyat (Fitzgerald translation)*

---

### read and write Reexamined

Before we go further, it will help to have a more precise notion of how the statements `read`, `readln`, `write`, and `writeln` work.

*arrow*

When a text file is opened, a location marker is placed somewhere in the file. For purposes of explanation, let us call this location marker an *arrow* and think of it as pointing to a file location that contains, or could contain, one character. This arrow tells where the next character to be read is or where the next character to be written will go. In the figures, we will shade the location pointed to by the arrow in order to make it more prominent.

*writing*

When a file is opened with the `rewrite` statement, the file is erased and the arrow is placed at the first location in the file. Every time the program writes a character, the character is written at the location of the arrow and then the arrow is advanced to the next location. Some sample code and its effect on the text file `TestFile` is shown in Figure 13.5. Notice that a second line is started when the `writeln` statement is executed. This is the only way to initiate a new line. There is no limit to the length of a line in a text file.

Also notice that in the example, the output to the file was written from the beginning of the file and continued on through the file. The arrow did not “back up.” When using a text file, the program cannot backspace and change a character. With text files, the moving arrow “writes; and, having writ, moves on,” much like the finger of fate described by Omar Khayyam. The only way to get the arrow back to a previous position is to use a `reset` or `rewrite` statement, and these do not allow the program to change a portion of the file. The `rewrite` statement will erase the entire contents of the file. The `reset` statement will move the arrow all the way back to the first character in the file and will only allow the program to read from the file. It will not allow the program to write to the file.









*lines*

*end-of-line  
marker*

In order to fully understand text files, we need to examine the notion of a *line* more carefully. Figure 13.5 is the way we normally think of lines, and text files are implemented so as to reflect this intuition. However, there are no physical lines in a text file. Instead, a text file is just one continuous stream of characters (rather like a ticker tape). The “end of a line” is indicated by inserting a special marker. This marker is a character of sorts, but it is not possible to see it on the screen. The computer can recognize it, though, and it is this character that indicates what we, and the Pascal manuals, call an



## The file is TestFile.

Program Action	File After the Action
open the file with rewrite	
write(TestFile, 'a')	a 
write(TestFile, 'b')	ab 
write(TestFile, 'c')	abc 
write(TestFile, 'd')	abcd 
writeln(TestFile)	abcd 
write(TestFile, 'e')	abcd e 
write(TestFile, 'f')	abcd ef 

**Figure 13.5**  
Writing to a text  
file, intuitive  
picture.

*end of a line.* In Figure 13.6 we have redone Figure 13.5, this time indicating the end of a line with this marker. In the diagram the marker is denoted `<eoln>`. The only way to insert an `<eoln>` marker in a text file is with a `writeln` statement.

An existing file is opened for reading with the `reset` statement. When a file is opened in this way, the arrow is placed at the first character of the first line of the file. Every time a character is read, the arrow is advanced to the next character. The situation is diagrammed in Figure 13.7. Notice that the `readln` statement moves the arrow to the beginning of the next line, and all the remaining characters on the old line are thus ignored. If the program runs out of characters on one line, there can be problems. Usually, the programmer must ensure that the arrow is explicitly moved to the next line by means of a `readln`. An exception is made for numbers. If the program is reading into a variable of type `integer` or `real` and there is no more data on the current line, it will automatically go to the next line.

The `<eoln>` symbol is a symbol in the file. However, it cannot be read into a variable of type `char` in the same way that other symbols can. If a program reads the

*reading*

*`<eoln>` symbol  
reads as a blank*

The file is TestFile.	
Program Action	File After the Action
Open the file with rewrite	█ ↑
write(TestFile, 'a')	a█ ↑
write(TestFile, 'b')	ab█ ↑
write(TestFile, 'c')	abc█ ↑
write(TestFile, 'd')	abcd█ ↑
writeln(TestFile)	abcd<coln>█ ↑
write(TestFile, 'e')	abcd<coln>e█ ↑
write(TestFile, 'f')	abcd<coln>ef█ ↑

**Figure 13.6**  
Writing to a text  
file, the “real”  
picture.

<coln> symbol into a variable, the variable will be filled with a blank; it will not be filled with the <coln> symbol. It is not possible to read it into a variable and then write it out as the <coln> symbol. If you wish to insert the <coln> symbol into a text file, you must do so with a `writeln` statement.

## eof and eoln

*eof*

When a program is reading from a text file, it is often helpful if it can detect the end of the file. Pascal provides a special boolean-valued function that does just that. The function `eof` is officially called the *end-of-file function*, but is usually pronounced by reading the letters “e-o-f”. It takes one argument, a file variable, and it returns `true` if the program is at the end of that file. More specifically,

`eof (<file variable>)`

evaluates to `true` if the arrow in the file indicated by <file variable> is past the last line in the file; otherwise, it evaluates to `false`.

As an example, the program in Figure 13.8 associates the file variable names `OldFile` and `NewFile` with the file string names `'OLD.TXT'` and `'NEW.TXT'`. After that, both we and the program will call the files by the names `OldFile` and `NewFile`. The program reads a list of numbers from the file `OldFile`, multiplies each number by 2, and copies the result into the file `NewFile`. The function `eof` is used to detect when all the numbers in the file have been read. Suppose the file `OldFile` contains the following when the program is run:

```
1
2
3
```

The body of the *while* loop in the program will then be executed three times. Each time, one of the following three lines will be written to the file `NewFile`:

```
2
4
6
```

---

**The file is called `TestFile`.**

---

Program Action	File After the Action		Value of X After Action
	Intuitive Picture	Real Picture	
open the file with reset	abcd ↑ ef	abcd<coln>ef ↑	?
read( <code>TestFile</code> , X)	abcd ↑ ef	abcd<coln>ef ↑	'a'
read( <code>TestFile</code> , X)	abcd ↑ ef	abcd<coln>ef ↑	'b'
readln( <code>TestFile</code> )	abcd ef ↑	abcd<coln>ef ↑	'b'
read( <code>TestFile</code> , X)	abcd ef ↑	abcd<coln>ef ↑	'e'

---

**Figure 13.7**  
Reading from  
a text file.



```

program TURBO_Doubler;
{Precondition: OLD.TXT contains a list of integers. Postcondition: OLD.TXT is unchanged;
NEW.TXT contains the numbers in OLD.TXT, each multiplied by 2.}
var Buffer: integer;
    OldFile, NewFile: text;
begin{Program}
    writeln('Program started. ');
    assign(OldFile, 'OLD.TXT');
    assign(NewFile, 'NEW.TXT');

    reset(OldFile);
    rewrite(NewFile);
    while not eof(OldFile) do
        begin{while}
            readln(OldFile, Buffer);
            Buffer := 2 * Buffer;
            writeln(NewFile, Buffer)
        end; {while}

    close(OldFile); close(NewFile);
    writeln('Numbers in OLD.TXT doubled');
    writeln('and results copied to NEW.TXT. ')
end. {Program}

```

**Figure 13.8**  
Program using  
eof.

At this point, the arrow in the file OldFile has moved beyond the third and last line. So eof (OldFile) evaluates to true, and the following boolean expression evaluates to false:

```
not eof (OldFile)
```

Since the controlling boolean expression now evaluates to false, the while loop is terminated.

*eoln* The function eoln is the same eoln that we have already used with input from the keyboard, but we will now see how to use it with other text files as well. It is similar to eof, except that it tests for the end of a line rather than the end of the entire file. In terms of the arrow discussed above, eoln can be explained as follows: When the arrow in the file specified by <file variable> is pointing to the end-of-line marker (what we have been denoting by <eoln> in the diagrams), then

```
eoln (<file variable>)
```

returns true; otherwise, it returns false. (Do not confuse eoln and <eoln>. <eoln> is a symbol in the text file. eoln is a function that tests to see if the arrow in a text file is pointing to <eoln>.)

In order to construct an example using eoln, suppose that a program contains the following declarations:

```

var OldFile, NewFile: text;
    Buffer: char;

```

If both files are opened properly and the arrow is at the start of a line, then the following loop will copy one line of text from OldFile into NewFile:

```
while not eoln(OldFile) do
  begin
    read(OldFile, Buffer);
    write(NewFile, Buffer)
  end
```

The effect of this loop on some sample file contents is shown in Figure 13.9. After the two characters on the line have been read, the arrow in OldFile is at the end-of-line marker, and so `eoln(OldFile)` becomes `true`. That causes the boolean expression for the `while` loop to become `false`, and so the loop terminates after the second iteration of the loop body. Typically, this loop would be followed by

```
readln(OldFile);
writeln(NewFile)
```

OldFile	NewFile	Buffer	eoln(OldFile)
ab<eoln>cd<eoln> ↑	█ ↑	?	false
then program executes <code>read(OldFile, Buffer)</code>			
ab<eoln>cd<eoln> ↑	█ ↑	'a'	false
then program executes <code>write(NewFile, Buffer)</code>			
ab<eoln>cd<eoln> ↑	a█ ↑	'a'	false
then program executes <code>read(OldFile, Buffer)</code>			
ab<eoln>cd<eoln> ↑	a█ ↑	'b'	true
then program executes <code>write(NewFile, Buffer)</code>			
ab<eoln>cd<eoln> ↑	ab█ ↑	'b'	true
then program executes <code>readln(OldFile)</code>			
ab<eoln>cd<eoln> ↑	ab█ ↑	'b'	false
then program executes <code>writeln(NewFile)</code>			
ab<eoln>cd<eoln> ↑	ab<eoln>█ ↑	'b'	false

**Figure 13.9**  
Copying a line  
from one text file  
to another.

This moves the arrow in each file to the beginning of the next line.

Recall that input from the keyboard is considered to be from a special file called `input`. The standard functions `eof` and `eoln` may be applied to this file `input`. Indeed, we have been using `eoln` with the file `input` since Chapter 7. If no argument is given, then the file is assumed to be `input`; that is, `eof` by itself is equivalent to `eof(input)` and, as we saw in previous chapters, `eoln` by itself is equivalent to `eoln(input)`. The end-of-line marker at the keyboard is the return key. The end-of-file marker at the keyboard varies from system to system. Since the exact details vary from system to system, using `eof` with the file `input` is usually more trouble than it is worth.

---

## Using a Buffer

The word “buffer” is frequently used in discussions about files and has a semitechnical meaning. A *buffer* is a location where some data is held on its way from one place to another. This is why we chose `Buffer` as the name of the character variable in the previous loop example. The program reads a character from one file into `Buffer` and then writes it from `Buffer` into the other file. The variable serves as a temporary location for one character.

---

## Pitfall

### Forgetting the File Variable in `eof` or `eoln`

There are a number of pitfalls associated with the special boolean functions `eof` and `eoln`. The simplest one of all is forgetting to include the file variable as an argument. For example, if you are testing for the end of the file `OldFile`, you must use `eof(OldFile)`. If you forget and instead use the unadorned `eof`, you are referring to the keyboard and not the file `OldFile`. This can be frustrating, since the program will compile without an error message. After all, the compiler has no way of telling that you did not mean to refer to the keyboard. However, problems will occur when the program is run. The program may stop and do nothing because it is waiting for input from the keyboard. If the program does not stop, then it is likely to produce an error message referring to an unexpected end of file. If your program is reading from the file `OldFile`, and instead of `eof(OldFile)` you mistakenly use `eof` to terminate the reading, then your program is in trouble. It is essentially guaranteed that the unadorned `eof` will not be true when your program reaches the end of the file `OldFile`, and so your program will attempt to read beyond the end of the file.

---



## Pitfall

### Use of `eoln` and `eof` with Numeric Data

Text files are designed for holding characters. When a program is reading or writing numbers, a type conversion is performed. The exact details of how a system handles numbers in text files will vary slightly from one installation to another. When you are dealing with numeric data, these details can cause the behavior of `eoln` and `eof` to vary in an unpredictable way from one system to another. The easiest way to avoid any problems when using `eof` with numeric data is to always use `readln` rather than `read`, and to avoid using `eoln` completely. When processing data of type `char`, Pascal has no such problems.

## Text Files as Parameters to Procedures

A text file can be a parameter to a procedure just as things of other data types can. There is, however, one qualification. Text file parameters must be *variable parameters*; they can never be value parameters.

As an example, we will design a procedure that has two text file parameters, one called `OldFile` and one called `NewFile`. The procedure will copy the contents of the file `OldFile` into the file `NewFile`. A precise definition of the task to be accomplished is

*example—  
copying  
a file*

*Precondition: OldFile has been opened with reset; NewFile has been opened with rewrite; but no reading or writing has taken place yet.*

*Postcondition: The contents of OldFile are unchanged; the contents of NewFile have been made the same as those of OldFile.*

The basic outline of the procedure is given in Figure 13.10.

In order to convert that piece of pseudocode into Pascal, all we need to do is to design some Pascal code for the informal instruction

Copy a line from `OldFile` to `NewFile`

This is exactly what we did in the section on `eoln`. The Pascal code we developed there is as follows:

```
while not eoln(OldFile) do
begin
    read(OldFile, Buffer);
    write(NewFile, Buffer)
end
```

If we now put together all the details, we obtain the procedure `Copy`, shown in Figure 13.11.

```

while not eof (OldFile) do
  begin{a line}

    {The arrows in OldFile and NewFile are
     both at the start of a line.}

    Copy a line from OldFile to NewFile

    {The arrows in OldFile and NewFile are
     both at the end of a line.}

    readln (OldFile); {Moves the arrow to the next line.}
    writeln (NewFile) {Inserts an end-of-line marker.}
  end {a line}

```

**Figure 13.10**  
Basic outline of  
the procedure in  
Figure 13.11.

```

procedure Copy (var OldFile, NewFile text);
{Precondition: OldFile has been opened with reset; NewFile has been
opened with rewrite; but no reading or writing has taken place yet.
Postcondition: The contents of OldFile are unchanged; the contents
of NewFile have been made the same as those of OldFile.}
var Buffer: char;
begin{Copy}
  while not eof (OldFile) do
    begin{a line and outer while}

      while not eoln (OldFile) do
        begin{inner while}
          read (OldFile, Buffer);
          write (NewFile, Buffer)
          {The current lines of NewFile and OldFile
           contain exactly the same thing up to (but not
           including) the positions of their arrows.}
        end; {inner while}
      {The arrow in OldFile is at the end of a line.}

      readln (OldFile);
      writeln (NewFile)
      {The arrows in both files are at
       the beginning of the next line.}
    end {a line and end outer while}
end; {Copy}

```

**Figure 13.11**  
Procedure  
with text file  
parameters.

---

## Pitfall

### Portability

The details of file handling vary from one implementation to another. Thus, a program that deals with files cannot be completely portable. Moving it from one implementation to another is likely to require rewriting part of the program. In order to make this rewriting task as easy as possible, all file handling should be isolated into self-contained procedures or at least easy-to-find isolated code. Then, if the program is moved to a new system, only these isolated sections need to be rewritten.

---

## Basic Technique for Editing Text Files

As we have already noted, there is no way to change part of a text file. The only way to write to a text file is to create a new file or to completely erase an old file. However, there is a way to get the effect of changing a file. In order to change part of a text file, a Pascal program must do something like the following:

1. Copy the entire contents of the given file into some temporary file, making changes as the copying is done.
2. Copy the entire contents of the temporary file back to the original file.

---

## TURBO Pascal

---

### Temporary Files

The basic technique we outlined for text editing requires a temporary file. This application is only one of many that require an extra, temporary file. In TURBO Pascal there are no special provisions for such temporary files. You must simply use some string name that is not the name of any file. In the next case study, the program asks the user to provide the name. Another alternative is to use some unlikely name, such as 'XXXX', but even in this case, it is a good idea to check with the user to see that there is no file with this unlikely string name.

After the program finishes its computation, we would like to delete any temporary files from the directory. This deletion can be done with the predefined TURBO Pascal procedure *erase*. The procedure takes one parameter which is the file variable name of a file and removes the file from the disk. If the file has been opened with either

*erase*



reset or rewrite, then it should be closed before it is deleted by the procedure erase. The use of temporary files and the procedure erase are illustrated in the following case study.

---

## TURBO Pascal—Case Study

---

### Editing Out Excess Blanks

#### Problem Definition

Suppose that you wish to write a program to edit excess blanks from a text file containing some ordinary English text. To be more precise, let us say that the program is to delete all initial blanks on a line and also is to compress all other strings of two or more blanks down to a single blank. For example, consider the lines below:

```
    The  Answer to the      question of Life,  
    the   Universe,          and Everything is:
```

They should be edited to look like

```
The Answer to the question of Life,  
the Universe, and Everything is:
```

#### Discussion

We will use the basic technique for editing text files that we outlined two sections ago. Let us use `DataFile` as the file variable name of the file to be edited. We will need one temporary file, which will have the file variable name `TempFile`. The basic outline of the program is as follows:

1. Open `DataFile` using `reset`; open `TempFile` using `rewrite`.
2. (`CleanBlanks`:) Copy `DataFile` into `TempFile`, but delete excess blanks as this is done. (That is, copy all characters except the unwanted blanks from `DataFile` to `TempFile`.)
3. Reopen `TempFile` using `reset`; reopen `DataFile` using `rewrite`.
4. (`Copy`:) Copy the contents of `TempFile` into `DataFile`.

We will implement subtasks 2 and 4 as procedures named `CleanBlanks` and `Copy`. The Pascal program, with some details still missing, is shown in Figure 13.12. The procedure `Copy` is the one in Figure 13.11. Thus, all that remains is to design the procedure `CleanBlanks` that will accomplish subtask 2.

Before designing the code for the procedure `CleanBlanks`, let us observe a few things about the program outline. Notice that the program in Figure 13.12 both reads from and writes to the same file. This is permitted as long as the reading and writing is not mixed. The program must first read from the file and then reopen it for the purpose

*mixing  
reading  
and  
writing*

```

program Edit;
{TURBO Pascal program that edits out excess blanks from the
text file specified by the user.}

var  DataFile, TempFile: text;
     FileName, SafeName: string[20];

procedure CleanBlanks(var DirtyFile, CleanFile: text);
{Precondition: DirtyFile has been opened with reset; CleanFile has
been opened with rewrite; but no reading or writing has taken place yet.
Postcondition: DirtyFile is unchanged; the contents of CleanFile are made
the same as those of DirtyFile except that superfluous blanks are deleted.}
.      .      .
.      .      .

procedure Copy(var OldFile, NewFile: text);
{Precondition: OldFile has been opened with reset; NewFile has been
opened with rewrite; but no reading or writing has taken place yet.
Postcondition: The contents of OldFile are unchanged; the contents
of NewFile have been made the same as those of OldFile.}
.      .      .
.      .      .

begin{Program}
  writeln('Enter name of file to be cleaned of blanks:');
  readln(FileName);
  assign(DataFile, FileName);

  writeln('I need a name which does not name any file. ');
  writeln('Enter any name that is not the name of a file: ');
  readln(SafeName);
  assign(TempFile, SafeName);

  reset(DataFile); rewrite(TempFile);
  CleanBlanks(DataFile, TempFile);

  reset(TempFile); rewrite(DataFile);
  Copy(TempFile, DataFile);

  close(DataFile); close(TempFile);
  erase(TempFile);

  writeln(FileName, ' cleaned of excess blanks. ')
end. {Program}

```

**Figure 13.12**  
**Program that edits**  
**a text file.**

of writing. It is also possible to write first and then read, but every change from reading to writing and every change from writing to reading requires that the file be reopened.

### CleanBlanks

We next design the algorithm and code for the one unfinished procedure. The procedure heading indicates what the procedure needs to accomplish:

```
procedure CleanBlanks (var DirtyFile, CleanFile: text);
{Precondition: DirtyFile has been opened with reset; CleanFile has
been opened with rewrite; but no reading or writing has taken place yet.
Postcondition: DirtyFile is unchanged; the contents of CleanFile are made
the same as those of DirtyFile except that superfluous blanks are deleted.}
```

### ALGORITHM

*negative  
thinking*

The procedure CleanBlanks can be very much like the procedure Copy. In fact, the only difference between CleanBlanks and Copy is that CleanBlanks will sometimes read a character and decide not to copy it to the second file. The algorithm is given in Figure 13.13.

We still need some way to tell when a character is an extra blank. For the moment, let us ignore the problem of initial blanks and concentrate only on strings of blanks within a line. We want to compress every string of two or more blanks to a single blank. One solution is to copy only the first blank and to consider the other blanks as extra blanks to be skipped over. In this approach a blank is extra (*not* copied) provided that

1. The character is a blank, and
2. The character that precedes it on the same line is also a blank.

This test requires that the program remember two characters instead of just one. Hence, we will use two variables of type `char` as buffer variables. One, called `Current`, serves the same purpose as the variable `Symbol` did in Figure 13.13. It will

```
while not eof(DirtyFile) do
begin{a line and outer while}
  while not eoln(DirtyFile) do
  begin{inner while}
    read(DirtyFile, Symbol);
    if Symbol is not an extra blank then
      write(CleanFile, Symbol);
    {The current lines of CleanFile and DirtyFile
    contain the same thing up to (but not including)
    the positions of their arrows, except that any
    excess blanks do not appear in CleanFile.}
  end; {inner while}
  {The arrow in DirtyFile is at the end of a line.}

  readln(DirtyFile);
  writeln(CleanFile)
  {The arrows in both files are at
  the beginning of the next line.}
end {a line and end outer while}
```



```

{Assumes that no line in DirtyFile starts with a blank.}
while not eof(DirtyFile) do
  begin{a line and outer while}
    initialize Last to something that will make the first iteration work;
    while not eoln(DirtyFile) do
      begin{inner while}
        read(DirtyFile, Current);
        if not( (Last = <blank>) and (Current = <blank>) ) then
          write(CleanFile, Current);
        Last := Current
        {The current lines of CleanFile and DirtyFile
         contain the same thing up to (but not including)
         the positions of their arrows, except that any
         excess blanks do not appear in CleanFile. Last
         contains the last character considered and possibly
         copied to CleanFile.}
      end; {inner while}
    {The arrow in DirtyFile is at the end of a line.}

    readln(DirtyFile);
    writeln(CleanFile)
    {The arrows in both files are at
     the beginning of the next line.}
  end {a line and end outer while}

```

**Figure 13.14**  
**First refinement of**  
**the algorithm for**  
**CleanBlanks.**

contain the current symbol, which either does or does not get copied to the second file. The other, called *Last*, will contain the previous character on the line being copied from. Therefore, the value of *Last* is just the previous value of *Current*. The algorithm refinement that uses these two variables is shown in Figure 13.14.

This is not yet a complete solution. It does not make sense for the first symbol of a line. To make it work for the first symbol of a line, recall that we want to copy the first symbol as long as it is not a blank. Hence, if we set *Last* equal to the blank symbol before we start each line, then this test works for the first symbol of a line as well. The complete, final code is shown in Figure 13.15.

*forcing the  
special case*

---

## Text Editing as a Programming Aid

A Pascal program is a piece of text and is stored in a text file. Hence, it can be edited by another program in the ways we have been describing. This can sometimes be a helpful programming aid. As an example, consider the task of tracing a program, which we discussed in Chapters 3 and 7. Tracing is a technique to aid in debugging programs. Specifically, it consists of inserting temporary `writeln` statements that output intermediate results. Once the program is debugged, we want to remove these

---

```

procedure CleanBlanks(var DirtyFile, CleanFile: text);
{Precondition: DirtyFile has been opened with reset; CleanFile has
been opened with rewrite; but no reading or writing has taken place yet.
Postcondition: DirtyFile is unchanged; the contents of CleanFile are made
the same as those of DirtyFile except that superfluous blanks are deleted.}
const Blank = ' ';
var Current, Last: char;
begin{CleanBlanks}
  while not eof(DirtyFile) do
    begin{a line and outer while}
      Last := Blank; {This ensures that a blank at the
start of a line will be deleted.}

      while not eoln(DirtyFile) do
        begin{inner while}
          read(DirtyFile, Current);
          if not( (Last = Blank) and (Current = Blank) ) then
            write(CleanFile, Current);
          Last := Current
          {The current lines of CleanFile and DirtyFile
contain the same thing up to (but not including)
the positions of their arrows, except that any
excess blanks do not appear in CleanFile. Last
contains the last character considered and possibly
copied to CleanFile.}
        end; {inner while}
      {The arrow in DirtyFile is at the end of a line.}

      readln(DirtyFile);
      writeln(CleanFile)
      {The arrows in both files are at
the beginning of the next line.}
    end {a line and end outer while}
  end; {CleanBlanks}

```

Figure 13.15

**Procedure to copy  
text with excess  
blanks omitted.**

extra `writeln` statements used for tracing. To aid us in finding and removing trace statements, or for that matter any other sort of temporary lines, we suggested marking them with a comment, as in the following sample of a temporary `writeln` statement:

```
{TEMP} writeln(Sum);
```

Suppose all our temporary statements are marked in the way we described. Then, to remove the temporary statements, all we need to do is locate those lines that begin with `{TEMP}` and delete them. This sort of tedious and uninteresting task is best left to

the computer. In Exercise 16 you are asked to write a program for the computer to do this editing. After you write the program, it would be a good idea to actually use the program as a programming tool from then on. It is not just a “toy problem.”

---

## Summary of Problem Solving and Programming Techniques

- Text files are used for storing data that is to remain in secondary storage after a program terminates. The data is stored as strings of symbols like the data displayed on the output screen.
- The text files that are manipulated by Pascal programs can also be read and changed using the editor.
- Any location, be it a variable or a text file or anything else, that holds data on its way from one place to another is called a *buffer*. Since it is not possible to move data directly from one file to another, a program that moves data between files must use a variable (or variables) as a buffer. The data is read from one file into the buffer and is then written out from the buffer to the other file.
- A text file may not be opened for reading and writing at the same time. Hence, in programs that edit a text file, the usual technique is to use an additional temporary file. The contents of the text file are copied into the temporary file, and the editing changes are made in the process of copying. After that, the edited version of the text is recopied back into the original file.
- One application of text editing is to edit the temporary trace statements out of a debugged Pascal program.
- The exact details of file handling will vary from one installation to another. Hence, file handling should be isolated into procedures in order to make any needed changes easy to carry out.
- When you are designing conditions for some program or algorithm action, it is sometimes clearer to think in terms of when the action should not take place, rather than thinking in terms of when it should take place.
- Once a general solution that applies to most cases has been found, it is often possible to force the solution to fit the remaining cases by setting some initial conditions or by making some other small changes.

---

In *theory*,  
there is no difference between *theory* and *practice*,  
but in *practice*,  
there is.

*Remark overheard at a computer science conference*

---



---

## Summary of Pascal Constructs

---

### Constructs Common to All Versions of Pascal

#### the type for text files

Syntax:

```
text
```

The Pascal type name for text files.

#### text file variables

Syntax:

```
var <file variable>: text;
```

Example:

```
var DataFile: text;
```

Declaration of a file variable of type `text`. These file variables are used as names for text files within a Pascal program.

#### reset

Syntax:

```
reset (<file variable>)
```

Example:

```
reset (File1)
```

Opens the text file specified by `<file variable>` for reading. `<file variable>` is a variable of type `text`. Reading starts at the first character of the first line of the text file and proceeds through the file. In TURBO Pascal a call to `reset` must be preceded by a call to `assign`.

#### rewrite

Syntax:

```
rewrite (<file variable>)
```

Example:

```
rewrite (File2)
```

Opens the text file specified by `<file variable>` for writing. `<file variable>` is a variable of type `text`. This always produces a blank file. If no file with `<file variable>` as its file variable name exists, then a blank one is created. If there already is a file associated with `<file variable>`, then the contents of that file are erased. In TURBO Pascal a call to `rewrite` must be preceded by a call to `assign`.

---

**write statement**

Syntax:

```
write (<file variable>, <argument list> )
```

Example:

```
write(DataFile, 'Hello', X, Y)
```

Just like using `write` to write to the screen, but when done this way, the items in `<argument list>` are written to the text file specified by `<file variable>`. `<file variable>` is a variable of type `text`. The file specified by `<file variable>` must have been opened with a call to `rewrite`. `<argument list>` is a list of variables and quoted strings separated by commas.

**writeln statement**

Syntax:

```
writeln (<file variable>, <argument list>)
```

Example:

```
writeln(DataFile, 'Hello', X, Y)
```

Same as the previous definition with the addition that it inserts an end-of-line marker in the text file. In other words, it causes any subsequent output to `<file variable>` to be written on the next line.

**read statement**

Syntax:

```
read (<file variable>, <variable list>)
```

Example:

```
read(DataFile, X, Y, Z)
```

Just like using `read` to read from the keyboard, but done this way the values are read from the text file specified by `<file variable>`. `<file variable>` is a variable of type `text`. The file specified by `<file variable>` must have been opened with a call to `reset`. The `<variable list>` is a list of variables separated by commas.

**readln statement**

Syntax:

```
readln (<file variable>, <variable list>)
```

Example:

```
readln(DataFile, X, Y, Z)
```

Same as the previous definition with the addition that it causes any subsequent read from the file to start at the beginning of the next line.

**end-of-file function**

Syntax:

```
eof (<file variable>)
```

Example:

```
eof (DataFile)
```

This is a boolean-valued function that returns `true` if all of the text file specified by `<file variable>` has been read. More precisely, it evaluates to `true` if the arrow (described in the chapter) has moved past the last line in the file. `<file variable>` is a variable of type `text`.

**end-of-line function**

Syntax:

```
eoln (<file variable>)
```

Example:

```
eoln (OldFile)
```

This is a boolean-valued function that returns `true` if all of the data on the current line of the file specified by `<file variable>` has been read. More precisely, it evaluates to `true` if the arrow (described in the chapter) is pointing to the end-of-line marker. `<file variable>` is a variable of type `text`.

---

## TURBO Pascal Constructs

**assign**

Syntax:

```
assign (<file variable>, <string name>)
```

Example:

```
assign (DataFile, 'DATA.TXT')
```

Associates the `<file variable>` name with the `<string name>` of a file. `<file variable>` is a variable of type `text`. `<string name>` is an expression of a *string* type. This is the first statement executed with the file. After a call to `assign` the file is referred to within the program by the name `<file variable>`.

**close**

Syntax:

```
close (<file variable>)
```

Example:

```
close (DataFile)
```

---



The procedure for closing the file named by the <file variable>. All files must be closed when a program is finished reading from or writing to the file. (If a file is already open and the program executes a `reset` or `rewrite` on the file, then the file is automatically closed before the `reset` or `rewrite` is executed. Hence, `close` need only be called once for each file, typically at the end of the program.)

### **erase**

Syntax:

```
erase (<file variable>)
```

Example:

```
erase (DataFile)
```

The disk file associated with <file variable> is deleted from the disk, and the directory is updated. If the file is open, it should be closed before `erase` is called.

---

## **More TURBO Pascal Constructs (Optional)**

### **rename**

Syntax:

```
rename (<file variable>, <string name>)
```

Example:

```
assign (DataFile, 'DATA.TXT');  
rename (DataFile, 'JUNK.TXT')
```

The file associated with the <file variable> is renamed <string name>. In the example, `DATA.TXT` is renamed `JUNK.TXT`. The file must not be open when it is renamed.

### **seekeoln**

Syntax:

```
seekeoln (<file variable>)
```

Example:

```
seekeoln (OldFile)
```

The same as `eoln` except that it skips over any blanks or tabs before testing for the end of a line. For example, if a program is reading from the text file with file variable name `OldFile` and nothing but blanks are left to read at the end of the current line, then `eoln(OldFile)` will return `false`, but `seekeoln(OldFile)` will return `true`.

---

**seekeof**

Syntax:

`seekeof (<file variable>)`

Example:

`seekeof (DataFile)`

The same as `eof` except that it skips over any blanks or tabs before testing for the end of a file. (Although `eof` can be used with other types of files, which we will introduce in Chapter 16, `seekeof` can be used only with text files.)

**append**

Syntax:

`append (<file variable>)`

Example:

`append (DataFile)`

This procedure is available on DOS systems, but may not be implemented in other versions of TURBO Pascal. The procedure opens the text file associated with the `<file variable>` for purposes of appending data to the end of the file. The old contents of the file are retained. The next `write` or `writeln` will cause text to be added after the old contents. A call to `append` need not be preceded by a call to `rewrite` or `reset`. However, it must be preceded by a call to `assign` just as with any other file opening. When writing to the file is completed, the file must be closed with a call to the procedure `close`. (`append` can be used only with text files; it cannot be used with the other types of files that will be introduced in Chapter 16.)

---

## Exercises

### Self-Test Exercises

5. What will be the contents of the file `ABE.TXT` after the following program is run with the following input from the keyboard:

```
program Exercise5;
var Abe: text;
    S: char;
begin{Program}
  writeln('Start input');
  assign(Abe, 'ABE.TXT');
  rewrite(Abe);
  while not eoln do
```

```

begin
    read(S);
    write(Abe, S)
end;
readln;
writeln(Abe);
readln(S);
writeln(Abe, S);
readln(S);
writeln(Abe, S);
writeln(Abe, 'The End. ');
close(Abe)
end. {Program}

```

### Input

Four score and seven years ago,  
our fathers brought forth upon  
this continent a new nation.

6. Suppose that the text file named by the file variable Sally contains the following:

```
abcdef<coln>ghijk<coln>lmnop<coln>
```

Suppose variables are declared as follows:

```

var Sally: text;
    L1, L2, Buffer: char;

```

What will be the output produced by the following, provided it is embedded in a complete program that opens Sally with reset and that declares variables as shown above?

```

readln(Sally, L1, L2); writeln(L1, L2);
while not eoln(Sally) do
begin
    read(Sally, Buffer);
    write(Buffer)
end;
writeln('Hi ')

```

7. Write a program to write the numbers 1 through 10 to a text file, one per line. Run the program and then look at the text file.
8. Write a program that reads the list of numbers from the text file of the previous exercise, computes their sum, and outputs the sum to the screen. The program should use eof to detect the end of the file.

### Interactive Exercise

9. Write a program that asks the user to enter the string name of a text file and then writes out the first line of text contained in that file.



### Programming Exercises

10. (This exercise may not be possible or may be very difficult if you are not using TURBO Pascal.) Write a program to help keep your disk free of old files. The program asks the user to enter a text file name. It then displays the first five lines of the text file and asks the user whether he or she would like to remove the file. If the file contains fewer than five lines, it displays the entire contents of the file. If the user indicates that the file should be removed, then the file is deleted from the disk using `erase`. This process is repeated until the user says that he or she wishes to quit the program.

11. Write a procedure that appends the contents of one text file to the end of another text file. The contents of the first file should be unchanged after the procedure is called.

12. Write a program that creates a table telling how to give change using quarters, dimes, and pennies. It should show the coins for all amounts from 1 to 99 cents. There should be a heading for the table. The program should output the table to a text file.

13. Write a program that outputs a table for converting from Celsius (centigrade) temperatures to Fahrenheit temperatures. Show all temperatures from minus 10 Celsius to 100 Celsius. A Celsius (centigrade) temperature  $C$  can be converted to an equivalent Fahrenheit temperature  $F$  according to the following formula:

$$F = (9/5)C + 32$$

The program should output the table to a text file.

14. Write a program that gives and takes advice on program writing. The program should open by writing a piece of advice to the screen. It should then ask the user to type in a different piece of advice. The next user of the program receives the advice typed in the last time the program was run. Be sure that the first person to run the program gets some advice.

15. The organization Professional Programmers for Purity in Pascal Programs has declared 'P' to be a dirty letter and has hired you to write a program to eliminate this letter from text files. Write a program that replaces all occurrences of the letter 'P' in a text file with the letter 'X'. The program should replace both upper- and lowercase letters, 'P' and 'p', with 'X'.

16. Write a program to delete all temporary lines from a text file containing a Pascal program. See the section entitled "Text Editing as a Programming Aid" for a discussion of this problem.

17. (This is an exploratory exercise for those who did the previous exercise.) In writing the program for Exercise 16, you may have added some temporary `write` or `writeln` statements. If you did not, go back now and add some. Now you have a copy of the program with lines that need to be deleted. Run this program on the text file containing the program; that is, "run the program on itself." The program will clean itself, so to speak. (Some systems may require that you have two copies of the program to do this, but most systems will let you do it with just one copy. In any event, you should have an extra copy of the program just in case some mistake causes the program to damage itself.)

---

18. Write a program that takes two lists of integers, each sorted from smallest to largest, and merges the numbers into a single larger list containing all of the numbers sorted from smallest to largest. The two lists are in two text files, and the merged list is placed in a third text file.

19. Write a program to generate personalized junk mail. The program will work with a text file that contains a letter in which the location of the name of the recipient is indicated by some special string of characters, such as '#name#'. The program will ask for the name from the keyboard, read it in, and then make a copy of the letter with the name inserted where indicated. The letter with the name inserted should be written to another text file.

20. Write a Pascal program which opens a text file containing a Pascal program and counts the number of assignment operators (`:` `=`) in the file. (Try running the program on the text file containing your Pascal program. Be sure to make a backup copy of the file with your program just in case something goes wrong and the program is damaged.)

21. Write a program that computes the average number of characters per word and the average number of words per sentence for the text in some text file.

22. Write a program that produces a list of all the words used in a text file as well as the number of times each word is used. The list should be output to the screen as well as to another text file. The list should be in alphabetical order. Do not forget to consider punctuation marks, such as periods and commas, when determining the ends of words. The program should treat upper- and lowercase letters as being the same. For example, 'Word' and 'word' should be treated as the same word.

23. Pascal allows either the pair '{ }' or the pair '(\* \*)' to enclose a comment. The reason for including the pair '(\* \*)' is that some systems do not have the symbols '{' and '}' available. In order to run a Pascal program written using '{ }' on a system that does not have these symbols, all occurrences of '{' need to be replaced with '(\*' and all occurrences of '}' need to be replaced with '\*)'. Write a program that changes the text file containing a Pascal program so that this substitution of comment delimiters is made.

24. (This exercise uses the optional sections of Chapter 8 covering random number generators.) Write a program to generate pseudorandom test data for other programs. The data is written to a text file that contains NumLines lines of NumPerL integer values per line. The values of NumLines and NumPerL, as well as a range of possible integer values, are to be read from the keyboard.

25. Books and newspapers contain text that is right-justified; that is, all lines are the same length. This is accomplished by copying as many words as possible onto one line and then adding extra blanks between the words so that the line is filled out to the prescribed line length. The line breaks in the original unjustified text are ignored in determining line lengths in the justified text. Write a program that produces right-justified text. The program will read from one file and write the right-justified text to another file. Use 80 characters, or whatever is convenient, as the line length.

26. (This exercise uses the optional sections of Chapter 8 covering random number generators.) Write a program to generate random English sentences. The program will

use two text files, one containing a list of nouns and one containing a list of verbs, and will use a random number generator to choose words from these files.

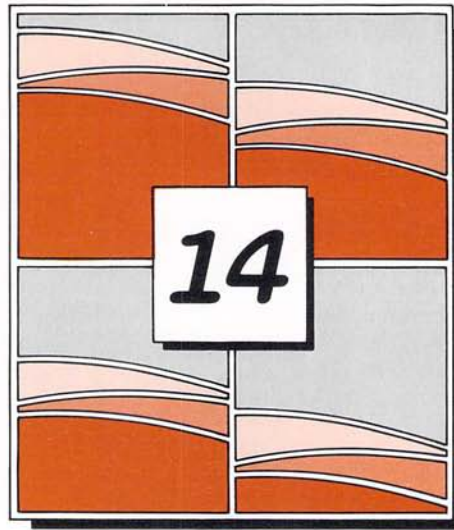
27. Enhance the program of the previous exercise so that it outputs a series of sentences that seem to be related. Do this by repeating the nouns and verbs used. Specifically, once a noun or verb is used, the program remembers it and uses it more often than the other words on the lists. To make the output seem even more reasonable, you might try grouping related words and have the program choose words related to those already chosen. Use your imagination in forming word groupings and in the other details of the algorithm.

28. Since the rules of grammar are not as rigid for poetry as they are for prose, it is usually easier to write a poetry-writing program that produces humanlike output than it is to produce a reasonable prose-writing program. Use the techniques discussed in the previous two exercises to write a program that outputs free verse poetry.

29. Design a version of Pascal in a foreign language of your choice. To do this, choose a fixed translation for each reserved word and standard identifier. Write a program that takes a text file containing a Pascal program and translates it into the foreign language version by replacing each reserved word and standard identifier with its translation. Then enhance your program so that it can also translate a foreign language program into English Pascal.

---





## *Problem Solving Using Recursion*

After a lecture on cosmology and the structure of the solar system, William James was accosted by a little old lady.

"Your theory that the sun is the center of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.

"And what is that, madam?" inquired James politely.

"That we live on a crust of earth which is on the back of a giant turtle."

Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.

"If your theory is correct, madam," he asked, "what does this turtle stand on?"

"You're a very clever man, Mr. James, and that's a very good question," replied the little old lady, "but I have an answer to it. And it is this: the first turtle stands on the back of a second, far larger, turtle, who stands directly under him."

"But what does this second turtle stand on?" persisted James patiently.

To this the little old lady crowed triumphantly. "It's no use, Mr. James—it's turtles all the way down."

## Chapter Contents

Case Study—A Recursive Function  
 A Closer Look at Recursion  
 Pitfall—Infinite Recursion  
 Stacks  
 Pitfall—Stack Overflow  
 Self-Test Exercises  
 Proving Termination and  
 Correctness for Recursive  
 Functions (Optional)  
 Case Study—A Simple Example of a  
 Recursive Procedure

Technique for Designing Recursive  
 Algorithms  
 Case Study—Towers of Hanoi—An  
 Example of Recursive Thinking  
 Recursive versus Iterative  
 Procedures and Functions  
 Case Study—Binary Search  
 Forward Declarations (Optional)  
 Summary of Problem Solving and  
 Programming Techniques  
 Exercises  
 References for Further Reading

*recursive  
 procedures  
 and  
 functions*

We have encountered a few cases of circular definitions that worked out satisfactorily. The most prominent examples are the definitions of certain Pascal statements. For example, the definition of a *while* statement says that it must contain another (smaller) statement. Since one possibility for this smaller statement is another *while* statement, there is a kind of circularity in that definition. The definition of the “*while* statement,” if written out in complete detail, will contain a reference to “*while* statements.” In mathematics these kinds of circular definitions are called “recursive definitions.” In Pascal, a procedure or function may be defined in terms of itself in the same way. To put it more precisely, a procedure or function declaration may contain a call to itself. In such cases, the procedure or function is said to be *recursive*. In this chapter we will discuss recursion in Pascal, and, more generally, we will discuss recursion as a programming and problem solving technique. We start with an example.

## Case Study

### A Recursive Function

In Chapter 8 we defined a function called `Power`, which computed integer powers of the form

$$x^n$$

That function was not recursive, but it will set the stage for a recursive version of the function. The function declaration is reproduced in Figure 14.1.

#### Problem Definition

Notice that the version of `Power` given in Figure 14.1 only works for nonnegative values of `N`. We will define a version of `Power` that computes powers for negative as well as nonnegative exponents. For a negative exponent  $-n$ , the value returned is, by definition,

$$x^{-n} = 1/x^n$$

#### Discussion

Our task is to extend the declaration so that the function also works for negative values of `N`. To get a feel for what is needed, let us consider a concrete case. Let us say that `X` is 3.0 and `N` is  $-2$ . We want

`Power(3.0, -2)`

to return the value  $(3.0)^{-2}$ . But that is equal to

$$1/(3.0)^2$$

```
function Power(X: real; N: integer): real;
{Returns X to the power N; Returns 1 when N equals 0.
  Precondition: N >= 0.}
var I: integer;
    Product: real;
begin{Power}
  Product := 1;
  for I := 1 to N do
    begin
      Product := Product * X
      {Product is X to the Power I.}
    end;
  Power := Product
end; {Power}
```

**Figure 14.1**  
Nonrecursive  
function  
declaration.



Negative powers are the same as positive powers in the denominator. Hence, if we know that the function `Power` returns the correct answer when `N` is positive, then we can calculate the correct value for `Power (3.0, -2)` by the expression

`1/Power (3.0, 2)`

### ALGORITHM

Now let us replace `-2` with `N`. This will produce the clause needed to extend the function declaration to accommodate negative values of `N`. The correct extra clause for the extended function declaration will be something equivalent to the following:

```
if N < 0 then
    Power := 1/Power (X, -N)
```

Remember, `N` is to take on negative values, such as `-2`. So `-N` will take on positive values, such as `2`. (The negative of a negative number is a positive number.)

recursive  
call

We can now put this all together to get a recursive declaration of `Power` that works for negative exponents as well as nonnegative ones. The declaration is shown in Figure 14.2. The function call

```
Power := 1/Power (X, -N)
```

is called a *recursive call* because it occurs inside of the declaration for `Power`.

Let us see what happens when this function is called with some sample values. First let us consider the simple expression

```
Power (3.0, 2)
```

When the function is called, the value of `X` is set equal to `3.0`, the value of `N` is set equal to `2`, and the code is executed. Since `2` is greater than zero, the *else* part is ignored. In this case, it is easy to see that the value returned is `9.0`.

```
function Power (X: real; N: integer): real;
{Returns X to the power N; Returns 1 when N equals 0.
Precondition: If N is negative, then X is not zero.}
var Product: real;
    I: integer;
begin {Power}
    if N >= 0 then
        begin {then}
            Product := 1;
            for I := 1 to N do
                Product := Product * X;
            Power := Product
        end {then}
    else {if N < 0 then}
        Power := 1/Power (X, -N)
    end; {Power}
```

**Figure 14.2**  
Recursive function  
declaration.

Next let us try a set of parameters that exercise the recursive part of the declaration. Let us determine the value of

Power (3.0, -2)

We proceed just as before. When the function is called, the value of  $X$  is set equal to 3.0, the value of  $N$  is set equal to -2, and the code is executed. Since -2 is less than zero, the *else* part is executed. Consequently, the value returned is the following (remember that since the value of  $N$  is -2, the value of  $-N$  is 2):

1/Power (3.0, 2)

We have already decided that the value of Power (3.0, 2) is 9.0. Hence, the value of Power (3.0, -2) is 1/9.0, which is approximately 0.11111111.

*example  
of a  
recursive  
call*

## A Closer Look at Recursion

The declaration of the function Power uses recursion. Yet we did nothing new or different in evaluating the function call Power (3.0, -2). We treated it just like any of the function calls we saw in previous chapters. We simply substituted the actual parameters for the formal parameters and then executed the code. When we reached the recursive call Power (3.0, 2), we simply repeated this process one more time.

The computer keeps track of recursive calls in the following way: When a function is called, the computer plugs in the actual parameters for the formal parameters and begins to execute the code. If it should encounter a recursive call, it temporarily stops its computation. This is because it must know the result of the recursive call before it can proceed. It saves all the information it needs in order to continue the computation later on and proceeds to evaluate the recursive call. When the recursive call is completed, the computer returns to and completes the outer computation. Figure 14.3 illustrates this for the example of the previous section. Notice that when a recursive call is encountered, the current computation is temporarily suspended and a second copy of the function declaration is used to evaluate the recursive call. When that is completed, the value returned by the recursive call is used to complete the suspended computation. In the example there are two levels of function calls. There may be several levels of recursive calls. The principle is the same no matter how many levels of recursive calls there are.

*how  
recursion  
works*

As a further example, let us rewrite the function Power one more time. Observe that for any number  $x$  and any positive integer  $n$ , the following relation holds:

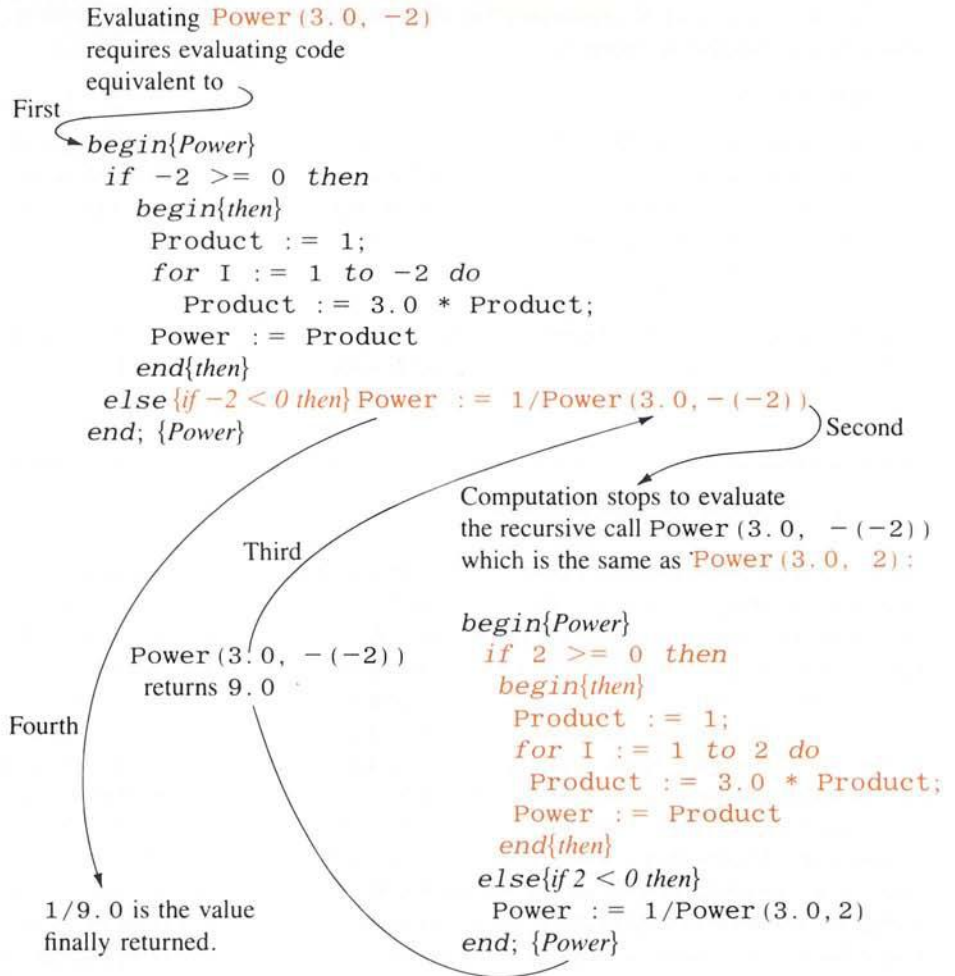
$$x^n = x(x^{n-1})$$

This means that an alternative way to define  $x^n$  is as follows:

The value is 1 when  $n = 0$ ;  
the value is  $x$  times  $x^{n-1}$  when  $n > 0$ ;  
the value is  $1/x^{-n}$  when  $n < 0$ .

*a highly  
recursive  
example*

*alternative  
ALGORITHM*



**Figure 14.3**  
Evaluation of a  
recursive function.

```

function POW(X: real; N: integer): real;
{Returns X to the power N; Returns 1 when N equals 0.
Precondition: If N is negative, then X is not zero.}
begin{POW}
  if N = 0 then
    POW := 1
  else if N > 0 then
    POW := X * POW(X, N-1)
  else {if N < 0 then}
    POW := 1/POW(X, -N)
end; {POW}

```

**Figure 14.4**  
A “highly”  
recursive function.



The Pascal version of a recursive function that computes in this way is given in Figure 14.4. To avoid confusion, we have used a slightly different name for this version of the function.

Evaluation of a recursive function such as POW can get fairly involved. Consider what happens when the following statement is executed:

```
Z := POW(2.0, 3)
```

The computer starts to evaluate POW(2.0, 3), but must stop to compute POW(2.0, 2). While computing POW(2.0, 2), it must stop to compute POW(2.0, 1). While computing POW(2.0, 1), it must stop to compute POW(2.0, 0). It can easily compute POW(2.0, 0) to be 1.0. It then returns and completes the computation of POW(2.0, 1) and determines that it is 2.0. It then uses this value to finish the computation of POW(2.0, 2), which it calculates to be 4.0. It then uses that value to complete the computation of POW(2.0, 3). The entire process is diagrammed in Figure 14.5. (To keep the figure uncluttered, we have simplified the code so that it uses the values of the actual parameters. Technically speaking, we should use the formal value parameters X and N as local variables and initialize them to these values. However, the idea is expressed more clearly by showing the values.)

*recursion  
within  
recursion*

## Pitfall

### Infinite Recursion

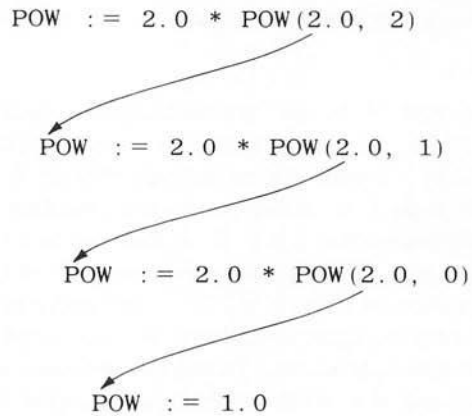
Pascal places no restrictions on how recursive calls are used in function and procedure declarations. However, in order for a recursive function declaration to be useful, it must be designed so that any call of the function must ultimately terminate with some piece of code that does not depend on recursion. The function may call itself, and that recursive call may call the function again. The process may be repeated any number of times. However, the process will not terminate unless eventually one of the recursive calls does not depend on recursion in order to return a value. The general outline of a recursive function declaration is as follows:

- One or more cases in which the value returned is computed in terms of simpler cases of the same function (i.e., using recursive calls).
- One or more cases in which the value returned is computed without the use of any recursive calls. These cases without any recursion are called *base cases* or *stopping cases*.

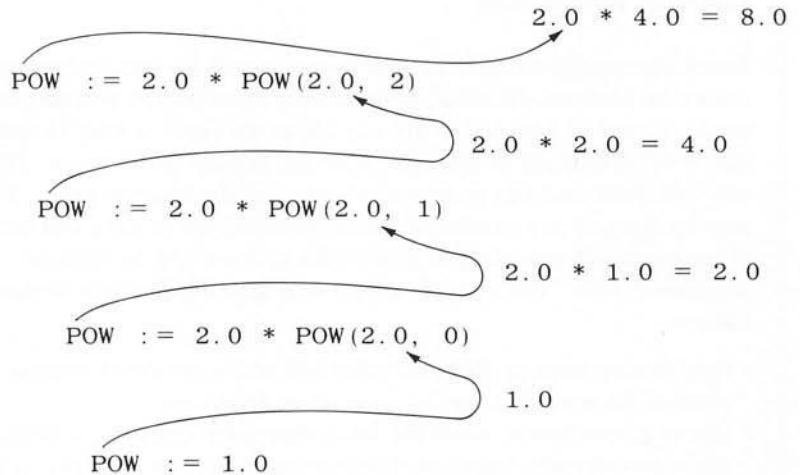
Often an *if-then-else* statement or a series of *if-then* statements determine which of the cases will be executed. A typical scenario is for the original function call to execute a case that includes a recursive call. That recursive call may in turn execute a case that requires another recursive call. For some number of times, each recursive call produces another recursive call, but eventually one of the stopping cases should apply. Every call of the function must eventually

*how  
recursion  
terminates*

Sequence of recursive calls



How the final value returned is computed



**Figure 14.5**  
Evaluating  
the recursive  
function call  
`POW(2.0, 3)`.

lead to a stopping case, or else the function call will never end because of an infinite string of recursive calls. (In practice, a call that includes infinite recursion will terminate abnormally rather than actually running forever.)

Every call of a recursive function should eventually lead to a stopping case. The most common way to ensure that a stopping case is eventually reached is to write the function so that some numeric quantity is decreased on each recursive call and to provide a stopping case for some “small” value such as zero. This is how we designed the function POW in Figure 14.4. Look back at Figure 14.5. To compute POW(2.0, 3), the function makes the following sequence of recursive calls: POW(2.0, 2), POW(2.0, 1), and POW(2.0, 0). The call to POW(2.0, 0) returns a value by executing the stopping case:

```
if N = 0 then
    POW := 1
```

That piece of code includes no recursive call. Hence, the code does eventually terminate; a value is returned for POW(2.0, 0), and the process works its way back to the original call, which terminates and returns a value for POW(2.0, 3).

In the previous example, the series of recursive calls eventually reached a call of the function that did not involve recursion (i.e., a stopping case). If, on the other hand, every recursive call produces another recursive call, then a call to the function will, in theory, run forever. In practice, such a function will run until the computer runs out of resources and terminates the program abnormally. Phrased another way, a recursive declaration should not be “recursive all the way down.” Otherwise, like the lady’s explanation of the universe, a call of the function will never end, except perhaps in frustration.

Examples of such infinite recursion are not hard to come by. The following is a syntactically correct Pascal function declaration that might result from an attempt to declare an alternative version of the function Power:

```
function RecPower(X: real; N: integer): real;
begin {RecPower}
    RecPower := 1/RecPower(X, -N)
end; {RecPower}
```

If this declaration is embedded in a program that calls this function, the compiler will translate it to machine code, and the machine code can be executed. Moreover, it even has a certain reasonableness to it. The relation

$$x^n = 1/x^{-n}$$

is true provided  $x$  is not zero. However, when called, this function will produce an infinite sequence of recursive calls. An attempt to evaluate RecPower(2.0, 3) will stop to evaluate RecPower(2.0, -3). That evaluation will in turn stop to evaluate an expression equivalent to RecPower(2.0, 3). That in turn will attempt to compute RecPower(2.0, -3). The process will proceed in a circle ad infinitum.

*example  
of infinite  
recursion*



## Stacks

To keep track of recursion, and a number of other things, most computer systems make use of a structure called a *stack*. A stack is a very specialized kind of memory structure that is analogous to a stack of paper sheets. In this analogy there is an inexhaustible supply of extra blank sheets. In order to place some information in the stack, it is written on one of these sheets of paper and placed on top of the stack of papers. To place more information in the stack, a clean sheet of paper is taken, the information is written on it, and this new sheet of paper is placed on the stack. In this straightforward way, more and more information can be placed on the stack. (This is a very common memory structure; a large number of office desks are organized in this fashion.)

Getting information out of the stack is also accomplished by a very simple procedure. The top sheet of paper can be read, and when it is no longer needed, it is thrown away. There is one complication: Only the top sheet of paper is accessible. In order to read, say, the third sheet from the top, the top two sheets must be thrown away. For this reason a stack is sometimes called a *last-in/first-out* memory structure.

Let us be a bit more precise about which pieces of paper are available to read and/or write on. In this analogy only the top sheet of paper on the stack is accessible. We will also have one other sheet that is available to work on. That extra sheet is not part of the stack, but it is still available. All sheets of paper in the stack other than the top one are not available. In order to access those sheets, the sheets above them must be thrown away.

*stacks  
and  
recursion*

Using a stack, the computer can easily keep track of recursion. Whenever a function is called, a new sheet of paper is taken. The function declaration is copied onto the sheet of paper, and the actual parameters are plugged in for the formal parameters. Then the computer starts to execute the body of the function declaration. When it encounters a recursive call, it stops the computation it is doing on that sheet in order to compute the value returned by the recursive call. But before computing the recursive call, it saves enough information so that, when it does finally determine the value returned by the recursive call, it can continue the stopped computation. This saved information is written on the sheet of paper and placed on the stack. A new sheet of paper is used for the recursive call. It writes a second copy of the function declaration on this new sheet of paper, plugs in the actual parameters, and starts to execute the recursive call. When it gets to a recursive call within the recursively called copy, it repeats the process of saving information on the stack and using a new sheet of paper for the new recursive call.

This process continues until some recursive call is completed and returns a value. When that happens, it takes the top sheet of paper off the stack. This sheet contains the partially completed computation that needs the value just returned. It is now possible to proceed with that computation. The process continues until the computation on the bottom sheet is completed. The value returned by that bottom computation is the value returned by the original function call. Depending on how many recursive calls are made and how the function declaration is written, the stack may grow and shrink in any fashion.

---

In Figure 14.6 we have redrawn the computation in Figure 14.5 showing how the stack behaves for this particular function call. Notice that the sheets in the stack can only be accessed in a last-in/first-out fashion, but that is exactly what is needed to keep track of recursive calls. The version currently being worked on is the one that was called by the version on top of the stack, and that is the one waiting for it to return a value.

Needless to say, computers do not have stacks of paper of this kind. This is just an analogy. The computer uses portions of memory rather than pieces of paper. The analogy is very exact, though. The contents of one of these portions of memory (“sheets of paper”) is called an *activation frame*. These activation frames are handled in this last-in/first-out manner, and the memory dedicated to holding these activation frames is called a *stack*.

Stacks are used for a number of things besides recursion. They are also used to keep track of local variables. Whenever any procedure calls another procedure, the computation of the calling procedure is suspended and an activation frame (“sheet of paper”) showing where it is in its computation is placed on the stack. Then a new activation frame is started for the procedure just called. If an identifier, say X, is local to the just-called procedure, then there may be values for X in both the new and the old activation frames. These are two different X’s: one local to the procedure just called and one local to some procedure with a wider scope. The computer knows they are different because they are in different activation frames. If you look back at Figure 5.11, you will see an illustration of how a stack can be used to keep track of the values of a variable identifier X when it is declared more than once. Although we did not call it a stack in Chapter 5, the illustrative boxes that contain the values of X show how a stack will change as the program is executed. The sequence of boxes shows the changes in stack contents. To simplify the figure, we have assumed that this stack does nothing other than keep track of the values of X. This is the same kind of stack as that used to keep track of recursive function calls. (Pascal does not use this exact mechanism to keep track of local variables, but uses a mixed strategy which combines this stack mechanism with a static approach. However, many programming languages do work in this fashion, and if you do not use global variables, then your Pascal program will behave as if the computer worked in this fashion.)

*activation  
frames*

*stacks  
and local  
variables*

## Pitfall

### Stack Overflow

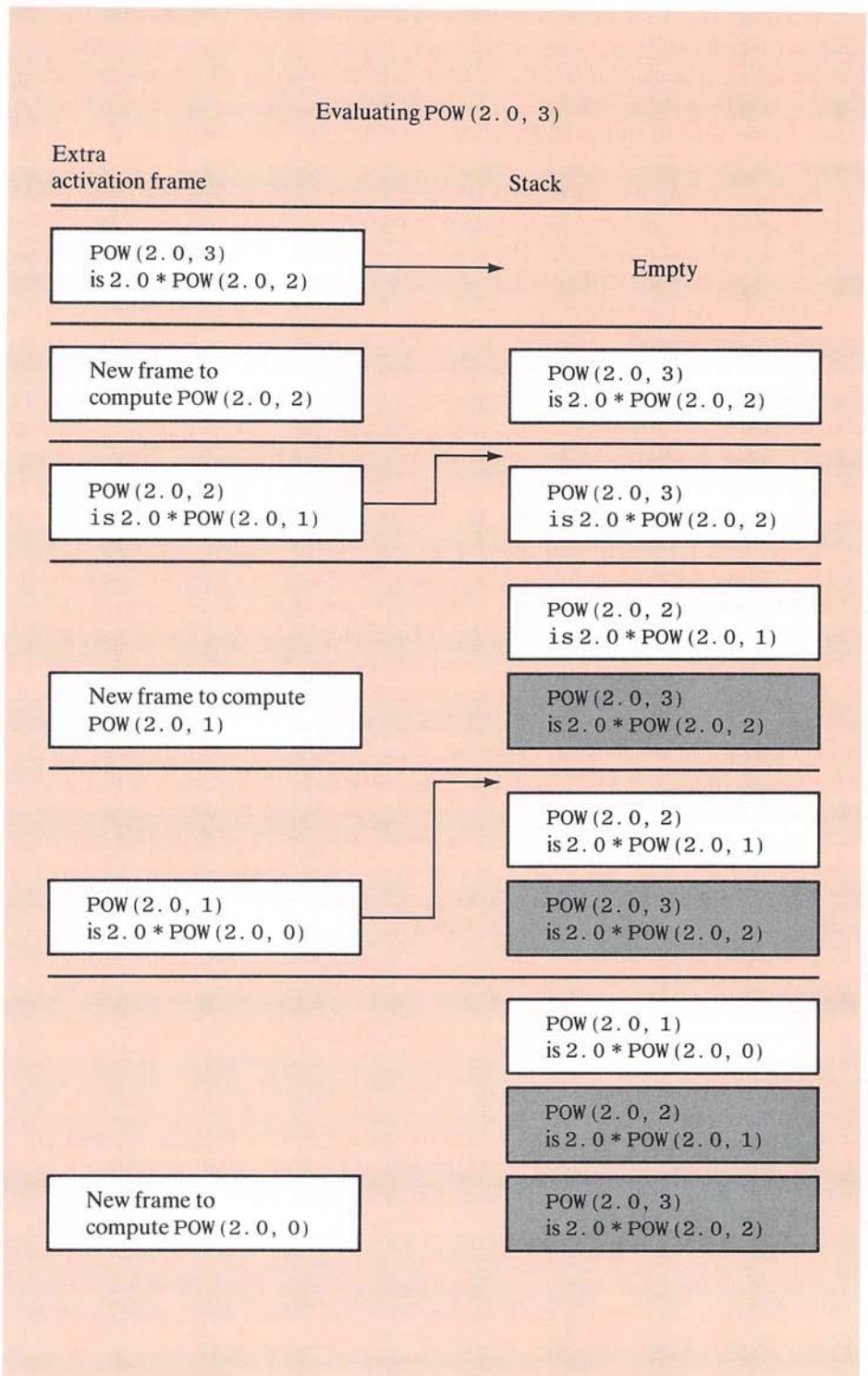
Many systems limit the size of the stack for various reasons. Suppose the stack is limited to 20 activation frames and that the function POW is declared as shown in Figure 14.4, and then suppose that the following statement is executed:

```
X := POW(2, 0, 30)
```

In that case the system will try to place approximately 30 activation frames on the stack. However, the stack is limited to 20 frames. The system cannot proceed

*stack  
overflow*





**Figure 14.6**  
Stack contents  
while evaluating a  
recursive function.



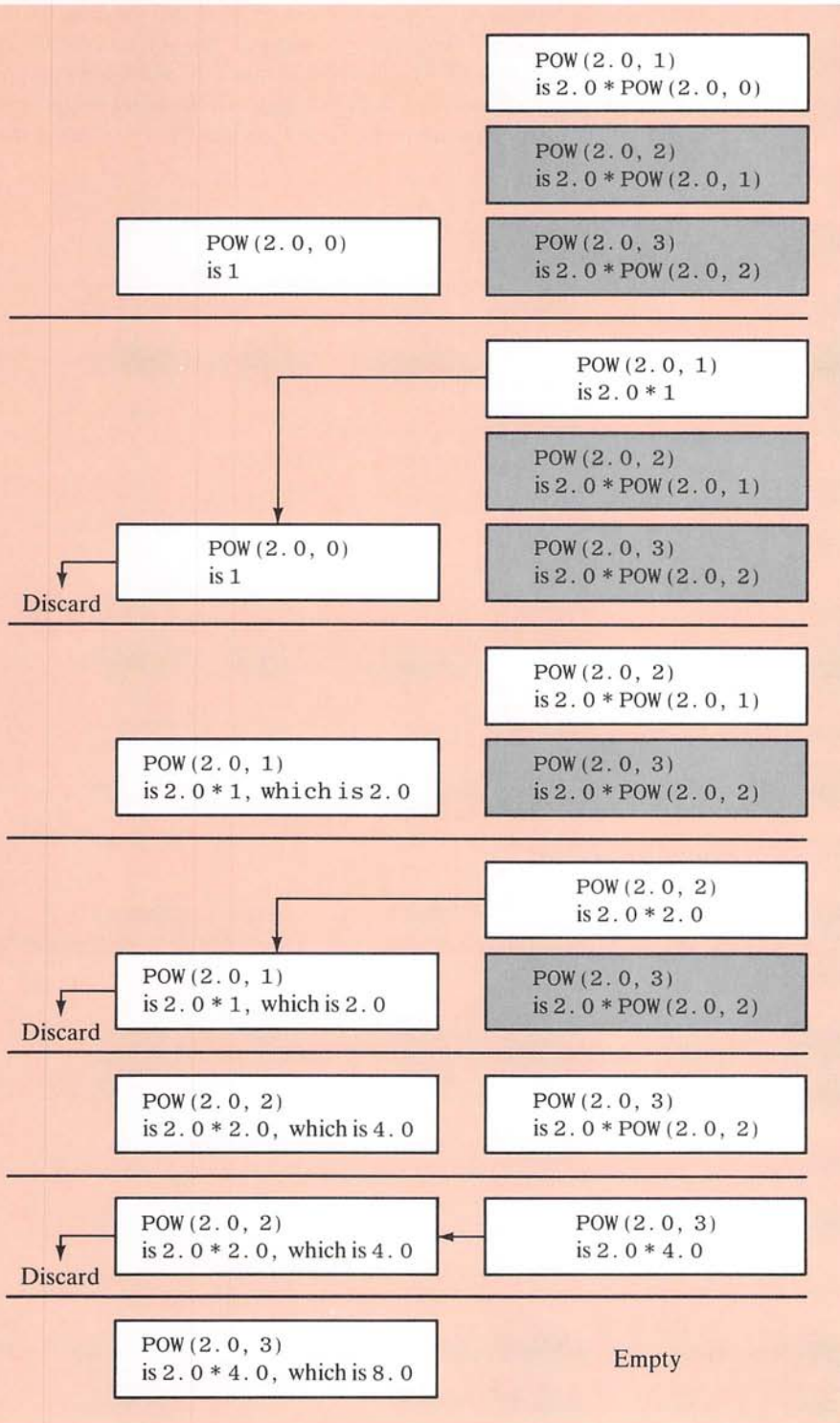


Figure 14.6  
(continued)

within the constraints imposed, the computation is aborted and an error statement is output. On many systems the error message will say *stack overflow*. A stack overflow simply means that the system tried to make the stack grow larger than is permitted. One common cause of stack overflow is infinite recursion. If a procedure or function is recursing infinitely, it will eventually try to make the stack exceed any stack size limit.

## Self-Test Exercises

1. What is the output of the following program?

```

program Exercise1(input, output);
function Mystery(N: integer): integer;
{Precondition: N >= 1}
begin{Mystery}
    if N = 1 then
        Mystery := 1
    else
        Mystery := N + Mystery(N - 1)
    end; {Mystery}
begin{Program}
    writeln(Mystery(3))
end. {Program}

```

2. Suppose that the function call in Exercise 1 is changed to

```
writeln(Mystery(0))
```

What will then be the output of the program?

3. What is the output of the following program? What well-known mathematical function is Rose?

```

program Exercise3(input, output);
function Rose(N: integer): integer;
{Precondition: N >= 0}
begin{Rose}
    if N = 0 then
        Rose := 1
    else
        Rose := N * Rose(N - 1)
    end; {Rose}
begin{Program}
    writeln(Rose(4))
end. {Program}

```

4. What is the output of the following program? Cabin is a fairly well-known mathematical function. You are likely to know of it or of a similar function.

```
program Exercise4(input, output);
function Cabin(N: integer): integer;
begin{Cabin}
  if N = 1 then
    Cabin := 0
  else
    Cabin := Cabin(N div 2) + 1
end; {Cabin}
begin{Program}
  writeln(Cabin(8))
end. {Program}
```

---

“I remembered too that night which is at the middle of the Thousand and One Nights when Scheherazade (through a magical oversight of the copyist) begins to relate word for word the story of the Thousand and One Nights, establishing the risk of coming once again to the night when she must repeat it, and thus to infinity.”

*Jorge Luis Borges, The Garden of Forking Paths*

---

---

## Proving Termination and Correctness for Recursive Functions (Optional)

Since a recursive function has the potential to go on without stopping, it is similar to a loop. As with a loop, the programmer has a responsibility to make certain that a function with a recursive call will terminate, provided the precondition was satisfied. Fortunately, this can be done by the same technique that we used to demonstrate that loops terminate. To prove that a recursive function terminates, it is enough to find a *variant expression* and *threshold* with the following properties:

*variant  
expression  
and  
threshold*

0. Whenever the function is called, it will either terminate or make a recursive call.
  1. There is some fixed amount such that, between one call of the function and any succeeding recursive call of that function, the value of the variant expression will decrease by at least this amount.
  2. If the function is called and the value of the variant expression is less than or equal to the threshold, then the function will terminate without making any recursive calls.
-



Condition 0 is included to take account of factors other than recursion. For example, if the function declaration consists of a loop followed by a recursive call of the function, then the loop must be shown to terminate by means of the techniques discussed in Chapter 7. That has nothing to do with recursion, but it can affect termination.

Conditions 1 and 2 have to do with recursion. To see that they, together with 0, guarantee termination, reason as follows: Suppose the three conditions hold. Since 0 is true, every call of the function will either terminate or produce a recursive call. Since 1 is true, every recursive call will decrease the variant expression. This means that either the function will terminate, which is fine, or else the variant expression will decrease until it reaches the threshold. But if condition 2 holds, then once the variant expression reaches the threshold, the function will terminate. That covers all the cases.

Interestingly enough, the preceding set of conditions is itself recursive. Conditions 0 and 2 include a test for termination, and it is termination for which we are testing. However, this is not a problem, since the conditions only discuss termination when there are no recursive calls of the function, and that kind of termination can be checked by the techniques we discussed in Chapter 7. A complete list of our conditions would also summarize those tests for termination.

As an example, consider the recursive function POW declared in Figure 14.4. The variant expression can be taken to be

abs (N) + 1 for negative values of N, and  
N for nonnegative values of N.

With a threshold of zero, the conditions 0 through 2 hold, and so we know that any call of the function will eventually terminate and return a value.

### induction

In addition to checking that a recursive function terminates, you should also check that it always returns the correct value. The usual technique for that is called *induction*. (If you have heard of mathematical induction, it may help to note that this is the same thing.) To show that a recursive function returns the correct value, all you need show is the following:

3. If the function returns without making any recursive calls, then it returns the correct value. (This is sometimes called the *base case*.)
4. If the function is called, and if all subsequent recursive calls return the correct value, then the original call will also return the correct value. (This is sometimes called the *inductive step*.)

By the *correct value* we mean whatever it is you want the function to return. That is part of the specification of the task of the function.

The conditions are numbered 3 and 4 to emphasize that they only ensure correctness if you know that the function calls always terminate. You must also ensure that conditions 0 through 2 hold in order to guarantee that a recursive function declaration performs its task as desired.

To complete our example, let us return to the function POW defined in Figure 14.4. To complete our demonstration that it performs as desired, we must show that 3 and 4 hold.

It is easy to see that condition 3 holds. The only way that the function can terminate without a recursive call is if the value of  $N$  is 0. In that case it returns 1, which is the answer we said we wanted. Any number to the power 0 is 1 by definition. (There is a reason for that definition, but that is a topic in algebra, not program verification.)

To see that condition 4 holds, we need only recall the algebraic identities

$$x^n = x(x^{n-1}),$$

and

$$x^n = 1/x^{-n}$$

## Case Study

### A Simple Example of a Recursive Procedure

#### Problem Definition

All our remarks about recursive functions apply equally well to recursive procedures. As a first example, consider the task of writing an integer to the screen with its decimal digits reversed. For example, the number 1234 should be output as

4321

#### Discussion

It is easy to decide what the first digit output should be, namely, the last digit of the number. If the number is 1234, then the first digit output is 4. Suppose we then delete that last digit from the number being processed. In the example, that would delete 4 from 1234, leaving the smaller number 123. The task remaining is then to output the digits of this smaller number in reverse order. Since this is a smaller version of the original problem, it is natural to use recursion here. We will design a recursive algorithm based on this strategy. The solution outline is as follows:

```

if the number is one digit long then
    write that digit
else
    begin
        write the last digit;
        remove the last digit;
        write the rest to the screen backwards
    end

```

*ALGORITHM*

In the example, 4 is the last digit and 123 is the rest. This algorithm is recursive because the last step performs the same task as the algorithm as a whole—writing a number backwards. It is routine to implement it as a recursive procedure. The imple-

*recursive  
solution*

**Program**

```

program ReverseTheDigits(input, output);
var Number: integer;

procedure WriteBackwards(Number: integer);
{Writes the decimal digits of Number to the screen
in reverse order. Precondition: Number >= 0.}
var LastDigit, TheRest: integer;
begin {WriteBackwards}
  if Number < 10 then
    write(Number:1)
  else
    begin {else}
      LastDigit := Number mod 10;
      TheRest := Number div 10;
      write(LastDigit:1);
      WriteBackwards(TheRest)
    end {else}
end; {WriteBackwards}

begin{Program}
  writeln('Enter a nonnegative whole number: ');
  readln(Number);
  writeln(Number, ' written backwards is: ');
  WriteBackwards(Number);
  writeln
end. {Program}

```

**Sample Dialogue**

**Figure 14.7**  
Program with a  
recursive  
procedure.

```

Enter a nonnegative whole number:
1066
    1066 written backwards is:
6601

```

mentation is shown in Figure 14.7, and execution for the parameter 1234 is shown in Figure 14.8. The arithmetic is straightforward. The last digit is always the remainder after division by 10; that is,

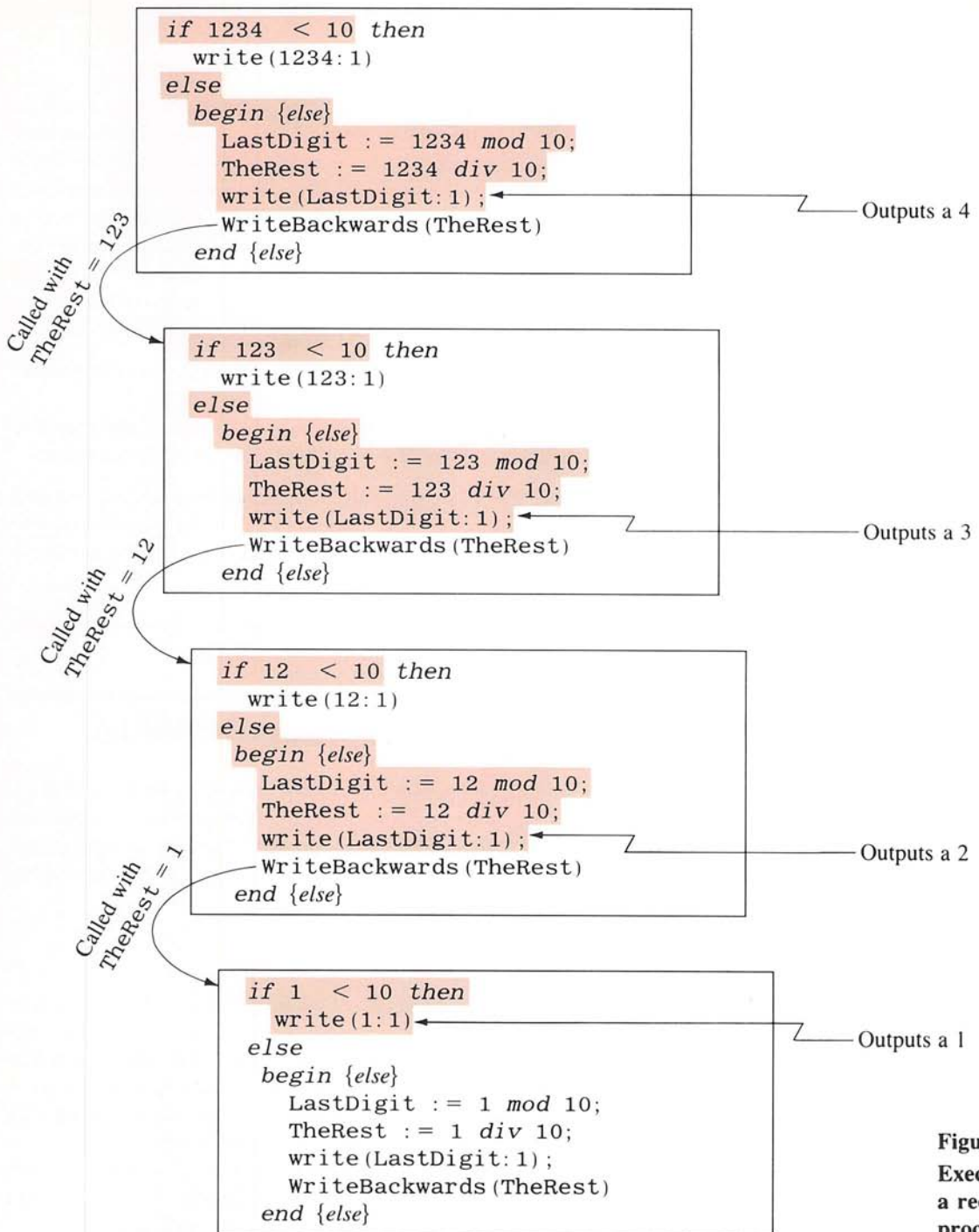
(the number) *mod* 10

For example, (1234 *mod* 10) is 4. “The rest” is just the quotient when the number is divided by 10; that is,

(the number) *div* 10

For example, (1234 *div* 10) is 123.





**Figure 14.8**  
Execution of  
a recursive  
procedure.

## Technique for Designing Recursive Algorithms

When one is designing a recursive procedure, the general technique is to try to divide the task into subtasks in such a way that one or more subtasks are smaller instances of the same problem. These smaller instances of the same problem will be the recursive calls. The sense in which they are smaller is a bit vague and depends on the particular problem, but the idea is that on each successive recursive call the tasks should become smaller and smaller until they have a simple solution that does not involve recursion. Hence, the algorithm and the final procedure declaration will include some cases that involve a recursive call and others that do not. As with recursive function declarations, the general outline of a recursive procedure declaration contains cases of two forms:

- One or more cases in which the procedure solves the problem in terms of simpler cases of the same problem (i.e., using recursive calls to itself).
- One or more cases in which the problem is solved without the use of any recursive calls. These cases without any recursion are called *base cases* or *stopping cases*.

As with recursive functions, every procedure call must ultimately lead to a stopping case; otherwise, the procedure call will produce infinite recursion. In the previous case study of reversing the digits in a number, the stopping case applies when the number is one digit long.

---

## Case Study

---

### Towers of Hanoi—An Example of Recursive Thinking

Recursion can be a very powerful programming tool. Sometimes a problem that appears to be difficult when tackled with any other programming technique can turn out to be very simple when thought of in terms of recursion. One dramatic example of this is provided by a children's game called Towers of Hanoi. As with any programming task, the first step is to understand the problem.

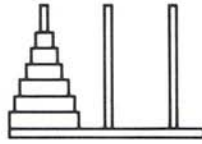
#### Problem Definition

The game consists of three pegs and a collection of rings that fit over the pegs, rather like phonograph records on a spindle. The rings are of different sizes. The initial configuration for a six-ring game is shown in Figure 14.9. Notice that the rings are stacked in order of decreasing size. A move consists of transferring a single ring from the top of one peg to that of another. The object of the game is to move all the rings from the first peg to the second peg. The difficulty is that you are never allowed to place a ring on top of one with a smaller radius. You do have one extra peg to temporarily hold rings, but the prohibition against placing a larger ring on a smaller ring applies to it as well as to the other two pegs. A solution for the case of three rings is given in Figure 14.9.

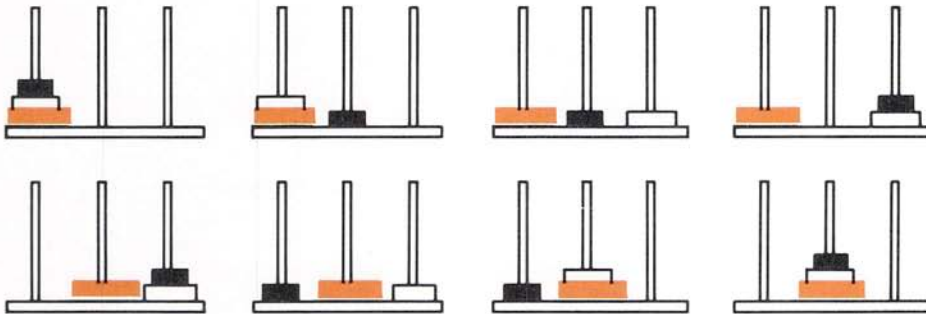
This game apparently has an impressive and long history. Legend has it that it was

---

Initial configuration for a six-ring game



Solution for a three-ring game

Figure 14.9  
Towers of Hanoi.

invented by God at the dawn of time and was given to man as one of the major tasks of humanity. The task of solving it eventually fell to the monks of a certain monastery in what was then an obscure Eastern village called Hanoi. The legend goes on to say that when the game is completed, the last task assigned to humanity will have been accomplished and that will mark the end of the world. The version God presented to mankind had 64 rings. So do not worry about trying it with three or four rings. (Besides, it cannot end the world unless the rings and pegs are made of stone and it is played in Hanoi.)

We want a procedure that solves the problem for any number of rings. Hence, there will be one parameter  $N$  that specifies the number of rings. The procedure cannot actually move rings, and so instead it will output instructions of the form “move a ring from peg  $x$  to peg  $y$ .” The list of instructions that is output will be instructions for solving the puzzle.

## Discussion

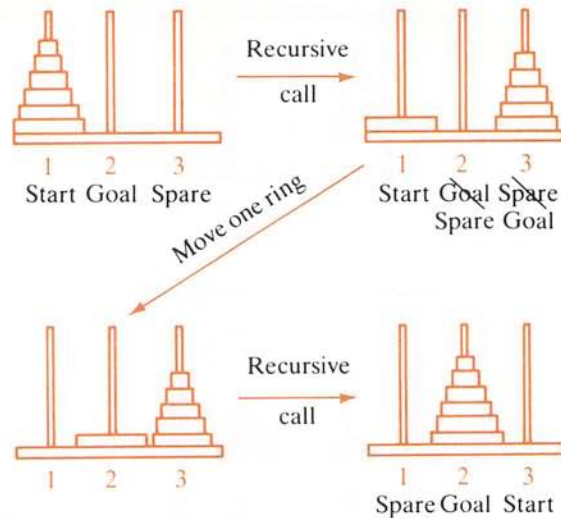
The game sounds simple enough, but a solution has not been easy to derive, at least not until very recently. It is easy to solve it for the case of three rings, but even four can take some thought. Ten rings boggle the mind. Sixty-four is almost unimaginable. To get the feel of it, try to solve it for the case of four rings. If you do, then try the six-ring case.

Although it is very difficult to solve by other means, the solution is almost trivial if you think recursively. The trick is to reduce the problem to a smaller one of the same

*parameters  
and output  
(first try)*

*thinking  
recursively*





**Figure 14.10**  
Idea of a recursive  
solution.

type. The trick is illustrated in Figure 14.10. First we move all but one ring onto the spare peg. We do this by a recursive call that uses peg 3 rather than peg 2 as the goal peg. Then we move the last ring to where it belongs. After that we move the rings from the original spare peg to peg 2 (the ultimate goal peg) by another recursive call.

All the pegs are identical, and so we can mix up their roles like this. Sometimes we make peg 3 the goal peg for a recursive call, even though our ultimate goal is to move the rings to peg 2. Moving the rings from the first peg to the second using the third as a spare is exactly the same problem as moving them from, say, the first to the third using the second as a spare.

Our analysis required that we change the roles of the various pegs. We want to move the rings to peg 2, but for the recursive calls we sometimes want to change the roles of the various pegs, using, for example, peg 3 in the role of the goal peg and thus moving some of the rings to peg 3. To allow this, we will use three more parameters, in addition to the parameter specifying the number of rings. The parameters will tell us which peg is the start peg, which is the goal peg, and which one is left over as a spare. For each call, these three parameters will be given values chosen from the integers 1, 2, and 3. The complete procedure heading will be the following, where the type `PegNumber` has been defined to be the subrange type 1 . . . 3.

*parameters  
(final form)*

```
procedure WriteMoves (N: integer; Start, Goal, Spare: PegNumber);
{Outputs the moves needed to move N rings
from Start peg to Goal peg, using Spare as the extra peg.}
```

For example, if we want to move rings from peg 3 to peg 1 using peg 2 as the spare, then we call the procedure with `Start` set equal to 3, `Goal` set equal to 1, and `Spare` set equal to 2.

**Program**

```

program Hanoi(input, output);
type PegNumber = 1 .. 3;
var N: integer;

procedure WriteMoves(N: integer; Start, Goal, Spare: PegNumber);
{Outputs the moves needed to move N rings
from Start peg to Goal peg, using Spare as the extra peg.}
begin{WriteMoves}
  if N = 1 then
    writeln('Move a ring from ', Start:1, ' to ', Goal:1)
  else
    begin{else}
      {Spare becomes the goal peg for a recursive call.}
      WriteMoves(N - 1, Start, Spare, Goal);

      writeln('Move a ring from ', Start:1, ' to ', Goal:1);

      {Spare becomes the start peg for a recursive call.}
      WriteMoves(N - 1, Spare, Goal, Start)
    end {else}
  end; {WriteMoves}

begin {Program}
  writeln('Enter the number of rings and');
  writeln('I'll explain how to play Towers of Hanoi. ');
  readln(N);
  writeln('To move ', N, ' rings');
  writeln('from peg 1 to peg 2, proceed as follows:');
  WriteMoves(N, 1, 2, 3);
  writeln('That does it.')
end. {Program}

```

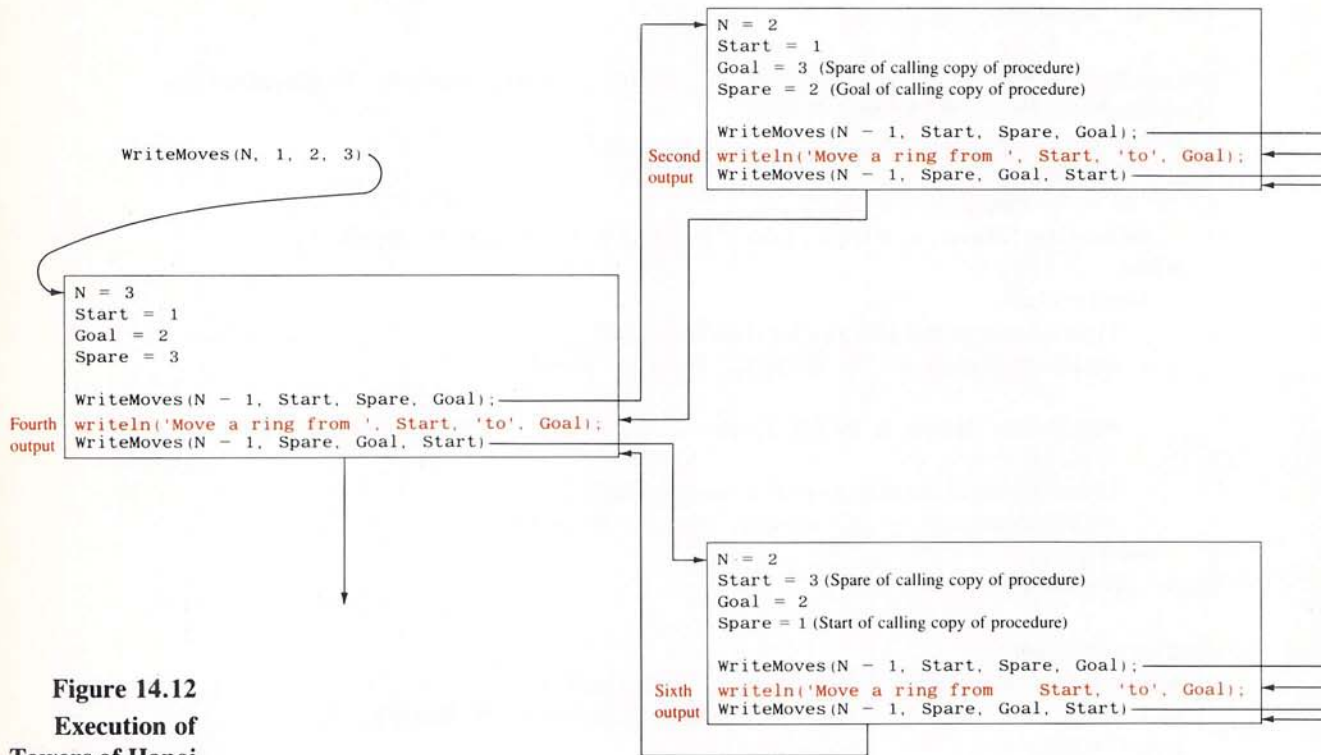
**Sample Dialogue**

```

Enter the number of rings and
I'll explain how to play Towers of Hanoi.
3
To move 3 rings
from peg 1 to peg 2, proceed as follows:
Move a ring from 1 to 2
Move a ring from 1 to 3
Move a ring from 2 to 3
Move a ring from 1 to 2
Move a ring from 3 to 1
Move a ring from 3 to 2
Move a ring from 1 to 2
That does it.

```

**Figure 14.11**  
**Program to play**  
**Towers of Hanoi.**



**Figure 14.12**  
Execution of  
Towers of Hanoi  
procedure.

### ALGORITHM

The algorithm is now easy to express:

```

if there is just one ring then
    instruct the user to move it from Start to Goal
else
    begin
        Output instructions for moving  $N - 1$  rings from Start to Spare;
        instruct the user to move one ring from Start to Goal;
        output instructions for moving  $N - 1$  rings from Spare to Goal
    end

```

A program with a complete recursive procedure to write out the moves is given in Figure 14.11. Figure 14.12 diagrams the sequence of recursive calls for the case of three rings.





Figure 14.12  
(continued)

## Recursive versus Iterative Procedures and Functions

Recursion is not absolutely necessary. In fact, many programming languages do not allow it. Any task that can be accomplished using recursion can also be done in some other way without using recursion. For example, Figure 14.4 contains a recursive function declaration. A nonrecursive version of that function is given in Figure 14.13. In such cases, the nonrecursive version typically uses a loop of some sort in place of recursion. For that reason, the nonrecursive version is usually referred to as an *iterative* version.

On many computer systems, the recursive version of POW given in Figure 14.4 will run slower and use more storage than the iterative version given in Figure 14.13. The

*iterative  
version*

```

function POW(X: real; N: integer): real;
{Returns the value of X to the N. Returns 1 whenever N is 0.
Precondition: If N is negative, then X is not zero.}
var I: integer;
    Product, Factor: real;
begin{POW}
    Product := 1.0;
    if N >= 0 then
        Factor := X
    else
        Factor := 1/X;
    for I := 1 to abs(N) do
        Product := Product*Factor;
    POW := Product
end; {POW}

```

**Figure 14.13**  
Iterative version of  
the function in  
Figure 14.4.

reason is that the recursive version uses a significant amount of time and storage to keep track of the recursive calls. Recall our discussion of how recursion is implemented using a stack. Suppose the recursive version is called to evaluate

POW(3.0, 10)

The computer will create about 10 stack frames in the process of evaluating this function call. This consumes an extra amount of time and memory. On the other hand, the iterative version does not do all this extra manipulating of the stack. It just performs 10 simple multiplications and returns the answer.

The POW example is typical. A recursively written function will usually run slower and use more storage than an equivalent iterative version. The difference in efficiency depends on how large the stack grows when the recursive version is used; that is, it depends on how long the string of recursive calls is. For example, the version of the function `Power` given in Figure 14.2 returns the same value as `POW`, is recursive, and yet is about as efficient as the iterative version of `POW`. This is because no call of `Power` ever produces a long string of recursive calls.

If efficiency is an important issue, it may make sense to avoid recursion. However, the efficiency issue is a subtle one. First of all, not all recursive declarations are equally inefficient. As discussed in the previous paragraph, the inefficiency introduced by the recursive call in the function `Power` of Figure 14.2 is negligible. Moreover, the recursion makes the code easier to read, since it reflects our normal manner of thinking about this computation. Also, recursion can sometimes make a procedure or function so much easier to understand that it would be foolish to avoid it. Consider the Towers of Hanoi procedure. If we did not think recursively, we might not have produced a solution at all. If we convert that procedure to an iterative procedure, it will be much more complicated and, as a result, will most likely contain bugs. No procedure can be considered efficient unless it gives the correct answers.

Finally, we should note that this discussion about efficiency assumes that recursive procedures are implemented with a stack using a method like the one we described. They need not be implemented in exactly that way. Some compilers will try to convert

a recursive function declaration to an iterative one before they translate the declaration into machine code. On one of these compilers, recursive and iterative versions of a function will probably be equally efficient.

---

## Case Study

---

### Binary Search

In this section we will develop a recursive procedure that searches an array to find out whether a given value is in the array. For example, the array may contain a list of the numbers for credit cards that are no longer valid, perhaps because the card has been stolen. A store clerk will need to search the list to see if a customer's card is still valid. In Chapter 9 (Figure 9.10) we discussed a simple method for searching an array by simply checking every array element. In Chapter 12 (Figure 12.5) we showed how the search could be made more efficient if the array is first sorted. In this section we will develop a method for searching a sorted array that is much faster than either of those two algorithms.

Let us call the array *A*. The indexes of the array *A* are the integers *First* through *Last*. *First* and *Last* are some specified integers, but their values are not relevant to the discussion. In order to make the task of searching the array easier, we will assume that the array is sorted. So if the array is *A*, then

$$A[\text{First}] \leq A[\text{First}+1] \leq \dots \leq A[\text{Last}]$$

When searching an array, we are likely to want to know both whether the value is in the list and (if it is) where it is in the list. For example, if we are searching for a credit card number, then the array index may serve as a record number. Another array indexed by these same indexes may hold a phone number or other information to use for reporting the suspicious card.

#### Problem Definition

We will design the procedure to use two variable parameters to return the outcome of the search. One parameter, called *Found*, will be of type *boolean* and will be set to *true* if the value is found. If it is found, then another parameter, called *Location*, will be set to the index of the value found. If we use *Key* to denote the value being searched for, the task to be accomplished can be formulated precisely as follows:

*Precondition:*  $\text{First} \leq \text{Last}$ ;  $A[\text{First}]$  through  $A[\text{Last}]$  are sorted into increasing order.

*Postcondition:* If *Key* is not one of the values  $A[\text{First}]$  through  $A[\text{Last}]$

then *Found* = *false*; otherwise  $A[\text{Location}] = \text{Key}$  and *Found* = *true*.

#### Discussion

Now let us proceed to produce an algorithm to solve this task. To do so it will help to visualize the problem in very concrete terms. Suppose the list of numbers is so long that it takes a book to list them all. This is, in fact, how invalid credit card numbers are

---



distributed to stores that do not have access to a computer. If you are a clerk and are handed a credit card, you must check to see if it is on the list and hence invalid. How would you proceed? Open the book to the middle and see if it is there; if not, and if it is smaller than the middle number, then work backward toward the beginning of the book; if it is larger than the middle number, work your way toward the back of the book. This idea produces our first draft of an algorithm:

**ALGORITHM**  
(first version)

```
Mid := approximate midpoint between First and Last;
if Key = A[Mid] then
    begin{found Key}
        Found := true;
        Location := Mid
    end {found Key}
else if Key < A[Mid] then
    search A[First] through A[Mid - 1]
else if Key > A[Mid] then
    search A[Mid + 1] through A[Last]
```

Since the searching of the shorter list is a smaller version of the very task that we are designing the algorithm to perform, it naturally lends itself to the use of recursion. The smaller lists can be searched with a recursive call to the algorithm.

Our pseudocode is a bit too imprecise to be easily translated into Pascal. The problem has to do with the recursive calls. There are two recursive calls shown:

```
search A[First] through A[Mid - 1]
search A[Mid + 1] through A[Last]
```

*more  
parameters*

In order to implement the recursive call, we need two more parameters. The recursive call specifies that a subrange of the array is to be searched. In one case, it is the elements indexed by First through Mid - 1. In the other case, it is the elements Mid + 1 through Last. The two extra parameters will specify the lower and upper bounds of the search. Let us call these two parameters Low and High. Using these parameters instead of First and Last, we can express the pseudocode more precisely as follows:

**ALGORITHM**  
(first refinement)

To search A[Low] through A[High], do the following:

```
Mid := approximate midpoint between Low and High;
if Key = A[Mid] then
    begin{found Key}
        Found := true;
        Location := Mid
    end {found Key}
else if Key < A[Mid] then
    search A[Low] through A[Mid - 1]
else if Key > A[Mid] then
    search A[Mid + 1] through A[High]
```

To search the entire array, the algorithm would be executed with Low set equal to First and High set equal to Last. The recursive calls will use other values for Low and High. For example, the first recursive call will set Low equal to First and High equal to the calculated value  $\text{Mid} - 1$ .

As with any recursive algorithm, we must ensure that our algorithm ends rather than producing infinite recursion. If the number is found on the list, then there is no recursive call and the process terminates, but we need some way to detect when the number is not on the list. On each recursive call the value of Low is increased or the value of High is decreased. If they ever pass and Low actually becomes larger than High, then we will know that there are no more indexes left to check and that the number is not in the array. If we add this test to our pseudocode, we obtain a complete solution, as shown in Figure 14.14.

Now we can routinely translate the pseudocode into Pascal. The result is shown in Figure 14.15. The procedure Search is an implementation of the above recursive algorithm. A diagram of how the procedure performs on a sample array is given in Figure 14.16.

Notice that the procedure Search solves a more general problem than the original task. Our goal was to design a procedure to search an entire array of type List. Yet the procedure will let us search any interval of the array by specifying the index bounds Low and High. This is a common phenomenon when designing recursive procedures. Frequently, it is necessary to solve a more general problem in order to be able to express the recursive algorithm. In this case, we only wanted the answer in the case where Low and High are set equal to First and Last. However, the recursive calls will set them to values other than First and Last.

The binary search algorithm is extremely fast compared to an algorithm that simply tries all array elements in order. In the binary search, we eliminate about half

*termination*

**ALGORITHM**  
*(final version)*

*solving a  
more general  
problem*

*efficiency*

```

if Low > High then
    Found := false
else
    begin{Low <= High}
        Mid := approximate midpoint between Low and High;
        if Key = A[Mid] then
            begin{found Key}
                Found := true;
                Location := Mid
            end {found Key}
        else if Key < A[Mid] then
            search A[Low] through A[Mid - 1]
        else if Key > A[Mid] then
            search A[Mid + 1] through A[High]
    end {Low <= High}

```

**Figure 14.14**  
**Pseudocode for**  
**binary search.**

```

program BinarySearch(input, output);
const First = 1;
      Last = 10;
type Index = First .. Last;
      List = array[Index] of integer;
var A: List;
    Key: integer;
    Found: boolean;
    Location: Index;

procedure Search(var A: List; Low, High, Key: integer;
                  var Found: boolean; var Location: Index);
{Precondition: A[Low] through A[High] are sorted into increasing order.
Postcondition: Key and A are unchanged; if Key does not equal one of A[Low]
through A[High], then Found = false, else Found = true and A[Location] = Key.}
var Mid: integer;
begin {Search}
  if Low > High then
    Found := false
  else
    begin{Low <= High}
      Mid := (Low + High) div 2;
      if Key = A[Mid] then
        begin{found Key}
          Found := true;
          Location := Mid
        end {found Key}
      else if Key < A[Mid] then
        Search(A, Low, Mid - 1, Key, Found, Location)
      else if Key > A[Mid] then
        Search(A, Mid + 1, High, Key, Found, Location)
      end {Low <= High}
    end; {Search}

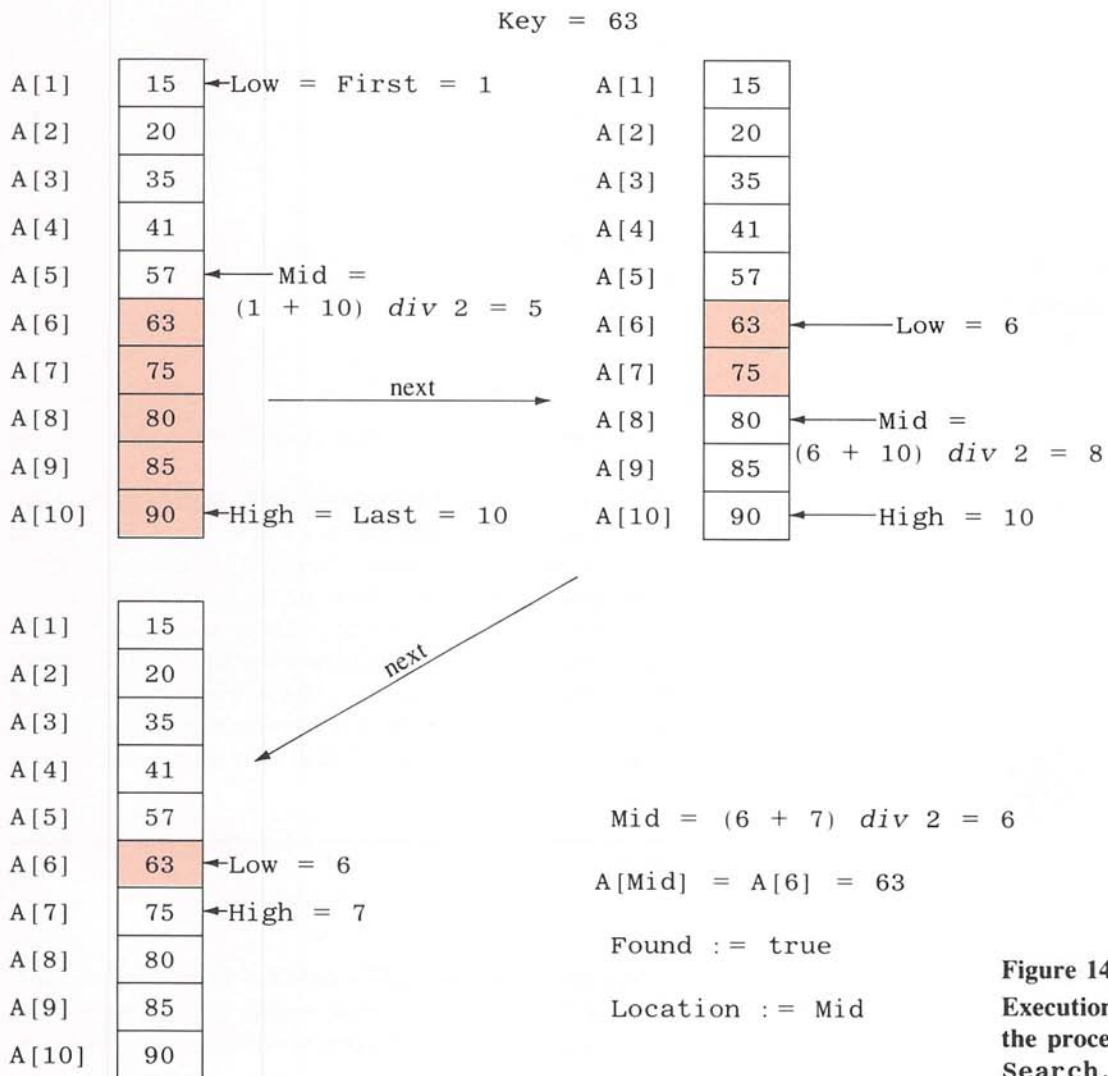
begin{Program}
  .
  .
  .
  This portion of the program contains some
  code to fill the array A. The exact
  details are irrelevant to the example.
  .
  .
  .
  writeln('Enter number to be located: ');
  readln(Key);
  Search(A, First, Last, Key, Found, Location);
  if Found then
    writeln(Key, ' is in location ', Location)
  else
    writeln(Key, ' is not in the array')
end. {Program}

```

**Figure 14.15**  
**Program with**  
**a recursive**  
**procedure for**  
**binary search.**



the array from consideration right at the start. We then eliminate a quarter and then an eighth of the array, and so forth. These savings add up to a dramatically fast algorithm. For an array of 100 elements, the binary search will never need to compare more than seven elements to the key. By contrast, either of the two simple search algorithms we have presented—Figures 9.10 and 12.5—could require as many as 100 comparisons, and on the average will require 50 comparisons to locate a key that is in the array. Moreover, the larger the array is, the more dramatic the savings will be. On an array with 1000 elements, the binary search will use at most 11 comparisons, compared to an average of 500 for the simple search algorithms. There are some cases for which the search algorithm in Figure 12.5 is faster in determining that a key is not in the array.



**Figure 14.16**  
Execution of  
the procedure  
Search.

```

procedure Search(var A: List; First, Last, Key: integer;
                var Found: boolean; var Location: Index);
{Precondition: A[First] through A[Last] are sorted into increasing order.
Postcondition: Key and A are unchanged; if Key does not equal
one of A[First] through A[Last], then Found = false
else Found = true and A[Location] = Key.}
var Mid, Low, High: integer;
begin {Search}
  Low := First;
  High := Last;
  Found := false; {so far}
  repeat
    Mid := (Low + High) div 2;
    if Key = A[Mid] then
      begin{found Key}
        Found := true;
        Location := Mid
      end {found Key}
    else if Key < A[Mid] then
      High := Mid - 1
    else if Key > A[Mid] then
      Low := Mid + 1
  until Found or (Low > High)
end; {Search}

```

**Figure 14.17**  
Iterative version of  
binary search.

However, these cases are not the most commonly occurring ones, and in other cases the binary search algorithm is dramatically faster.

*iterative  
version*

Any recursive procedure can be replaced by an iterative procedure that accomplishes the same task. In some cases, the iterative procedure may be more efficient, and if efficiency is a major issue, you may want to convert a recursive procedure to an iterative one. An iterative version of the binary search algorithm is given in Figure 14.17. On some systems it will run more efficiently than the recursive version. The algorithm for the iterative version was derived by mirroring the recursive version. In the iterative version, the local variables `Low` and `High` mirror the roles of the parameters of the same names in the recursive version. As this example illustrates, it often makes sense to derive a recursive algorithm even if you expect to later convert it to an iterative algorithm.

---

## Forward Declarations (Optional)

Normally, you declare a procedure or function before the place where it is first used. However, there is a way around this rule. If you wish to declare a procedure after the declaration of some other procedure that uses it, you can, provided you warn the com-

---

**Program**

```

program Test(input, output);
{Tests the mutually recursive procedures GetAnswer and Reject.}
var Ans: char;

procedure Reject(var Ans: char); forward;
{Outputs a message saying Ans is not an acceptable
answer and then calls the procedure GetAnswer(Ans).}

procedure GetAnswer(var Ans: char);
{Sets the value of Ans to 'Y' or 'N' depending on what the user
types in. Repeats the process until the user types in one of these
two characters. Mutually recursive with the procedure Reject.}
begin{GetAnswer}
    writeln('Answer cap Y for Yes or cap N for No. ');
    readln(Ans);
    if not (Ans in ['Y', 'N']) then
        Reject(Ans)
end; {GetAnswer}

procedure Reject {(var Ans: char)};
begin{Reject}
    writeln(Ans, ' is not an acceptable response. ');
    GetAnswer(Ans)
end; {Reject}

begin{Program}
    writeln('This is a test. ');
    GetAnswer(Ans);
    writeln('The value of Ans is ', Ans);
    writeln('That ends the test. ')
end. {Program}

```

**Sample Dialogue**

```

This is a test.
Answer cap Y for Yes or cap N for No.
y
y is not an acceptable response.
Answer cap Y for Yes or cap N for No.
OK
O is not an acceptable response.
Answer cap Y for Yes or cap N for No.
Y
The value of Ans is Y
That ends the test.

```

**Figure 14.18**  
**(Optional)**  
**Example of a**  
**forward**  
**declaration.**



piler by including a *forward declaration* before the first location where the procedure is called. A forward declaration consists of the procedure heading followed by the identifier *forward* and terminated by a semicolon. For example,

```
procedure Reject (var Ans: char); forward;
```

The procedure declaration can then be placed anywhere after the forward declaration. Since the formal parameter list is given in the forward declaration, it is not given again when the procedure is declared. (However, it is a good idea to include the parameter list in a comment.) Functions as well as procedures can be given forward declarations in this way.

*mutual  
recursion*

The program in Figure 14.18 requires a forward declaration. In that program the procedures `GetAnswer` and `Reject` each include a call to the other. Such a phenomenon is called *mutual recursion*. Forward declarations can also be used for less essential reasons, such as for putting all the most important procedures together in one place.

---

## Summary of Problem Solving and Programming Techniques

- If a problem can be reduced to smaller instances of the same problem, then a recursive solution is likely to be easy to find and easy to implement.
  - A recursive algorithm for a function or procedure declaration normally contains two kinds of cases: one or more cases that include a recursive call and one or more *stopping* cases in which the problem is solved without the use of any recursive calls.
  - When writing recursive procedures or functions, always check to see that the procedure will not produce infinite recursion.
  - When you are designing a recursive procedure to solve a task, it is often necessary to solve a more general problem than the given task. This may be required in order to allow for the proper recursive calls, since the smaller problems may not be exactly the same type of problem as the given task. For example, in the binary search problem, the task was to search an entire array, but the recursive solution is an algorithm to search any portion of the array (either all of it or a part of it).
- 

—————  
To iterate is human, to recurse divine.

Anonymous  
—————

---

---

## Exercises

### Self-Test Exercises

5. What is the output of the following program?

```
program Test2(input, output);
procedure Cheers(N: integer);
begin{Cheers}
  if N = 1 then
    writeln('Hurray')
  else
    begin{else}
      write('Hip ');
      Cheers(N - 1)
    end {else}
end; {Cheers}
begin{Program}
  Cheers(3)
end. {Program}
```

6. Write an iterative version of the function *Mystery* declared in Exercise 1.  
7. Write an iterative version of the function *Rose* declared in Exercise 3.  
8. Write a recursive procedure that has one parameter which is a positive integer and that writes out that number of asterisks '\*' to the screen.

### Interactive Exercises

9. Take a pad of paper to use as the inexhaustible supply of paper for a stack. Simulate the following procedure call using a stack of real paper. First, use the one-word sentence *Hi*. Next, use a sentence of your own choice that is about 10 characters long. After that, type up the program and run it.

```
program ReverseInput(input, output);
procedure RecReadWrite;
const Period = '.';
var Letter: char;
begin{RecReadWrite}
  read(Letter);
  if Letter <> Period then
    RecReadWrite;
  write(Letter)
end; {RecReadWrite}
begin{Program}
  writeln('Type in a sentence ending with a period.
  RecReadWrite;
  writeln
end. {Program}
```

---

10. Get hold of a real Towers of Hanoi game. Run the program in Figure 14.11 and follow its instructions for playing the game. Also simulate the program using a stack of paper, as in the previous exercise. The stack simulation is worth doing even if you do not have the game available.

### Programming Exercises

11. Write a recursive function with one argument *N* of type `integer` that returns the *N*th Fibonacci number. See Exercise 28 in Chapter 7 for the definition of Fibonacci numbers.

12. The formula for computing the number of ways of choosing *r* different things from a set of *n* things is the following:

$$C(n, r) = \frac{n!}{r! (n - r)!}$$

*n*! is the factorial function ( $n! = n*(n-1)*(n-2)*\dots*1$ ). Discover a recursive version of the above formula and write a recursive Pascal function that computes the value of the formula.

13. Write a recursive procedure that has as arguments an array of characters and two bounds on array indexes. The procedure should reverse the order of those entries in the array whose indexes are between the two bounds. For example, if the array is

A[1] = 'A' A[2] = 'B' A[3] = 'C' A[4] = 'D' A[5] = 'E'

and the bounds are 2 and 5, then after the procedure is run the array elements should be

A[1] = 'A' A[2] = 'E' A[3] = 'D' A[4] = 'C' A[5] = 'B'

Embed the procedure in a program and test it.

14. Write an iterative version of the procedure in the previous exercise. Embed it in a program and test it.

15. Write a recursive procedure to sort an array of integers into ascending order using the following idea: First place the smallest element in the first position, and then sort the rest of the array by a recursive call. (This is a recursive version of the selection sort algorithm discussed in Chapter 9.)

16. Write a procedure that takes two parameters that are arrays of integers of the same size and one parameter that is an array of integers of twice that size. The procedure assumes that the two smaller arrays are sorted and copies their contents into the larger array. It does so in such a way that the integers in the larger array are also sorted. Embed the procedure in a program in order to test it.

17. Use the ideas of the previous exercise to design a recursive sorting procedure that works along the following general lines: The array is divided in half. Each half is sorted by a recursive call and then the two halves are merged into a single sorted array. Embed the procedure in a program and test it.

18. Write an iterative version of the procedure in the previous exercise.



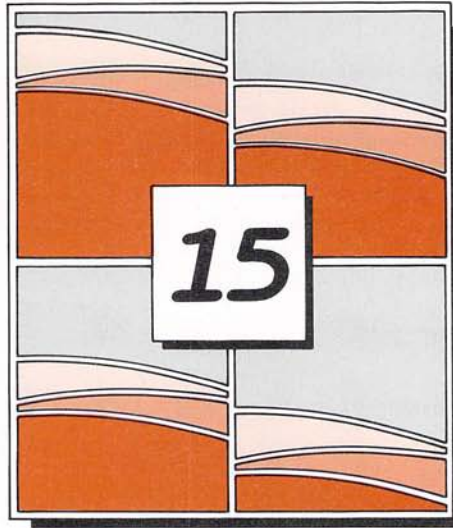
19. Write an iterative version of the procedure `WriteMoves` from the Towers of Hanoi program (Figure 14.11).
20. Write a set of procedures for using an array of characters as a stack. There should be one procedure to add a character to the stack, one to remove a character, and one to read the “top” character on the stack.
21. Write a recursive procedure that takes as input an array of characters and outputs all permutations of the characters in the array. The array can hold a maximum of five characters, but need not be full. One other parameter tells how many array elements are being used.
22. Find a formula for the number of times a ring is transferred from one peg to another in the Towers of Hanoi game with  $n$  rings. Compute the values for  $n$  equal to 5, 10, and 20.
23. Rewrite the Towers of Hanoi procedure so that it draws a stylized picture of the game being played. The output should be a series of pictures showing the game configuration after each ring is moved.
24. A *pretty-print* program takes a program, which may not be indented in any particular way, and produces a copy with the same program indented so that *begin/end* pairs line up, with inner pairs indented more than outer pairs, and so that *if-then-else* statements are indented, with the *if* and *else* lined up and with comments lined up and so forth. Write a program that reads a Pascal program from one text file and produces a pretty-print version of the program in a second text file. To make it easier, simply do this for the body of the program, ignoring the declarations, and assume that all substatements of complex statements (other than compound statements themselves) are compound statements enclosed in *begin/end* pairs. To make it harder, add any or all of the features omitted from the easy version.

---

## References for Further Reading

- P. Helman and R. Veroff, *Intermediate Problem Solving and Data Structures*, 1986, Benjamin/Cummings, Menlo Park, Ca., Chapters 4 and 5.
- E.S. Roberts, *Thinking Recursively*, 1986, John Wiley & Sons, New York.
- J.S. Rohl, *Recursion via Pascal*, 1984, Cambridge Computer Science Texts 19, Cambridge University Press, Cambridge and New York, Chapter 1.
-





## *Solving Numeric Problems*

I confess that I cannot recall any case within my experience which looked at first glance so simple, and yet which presented such difficulties.

*Sir Arthur Conan Doyle,  
(Sherlock Holmes) The Man with the Twisted Lip*



## Chapter Contents

A Hypothetical Decimal Computer  
Binary Numerals (Optional)  
Machine Representation of Numbers  
in Binary (Optional)  
Extra Precision (Optional)  
Self-Test Exercises  
Pitfalls—Sources of Error in real  
Arithmetic  
Pitfall—Error Propagation

Case Study—Series Evaluation  
Case Study—Finding a Root of a  
Function  
Summary of Problem Solving and  
Programming Techniques  
Summary of Terms  
Exercises  
References for Further Reading

Most of the computing done by scientists and engineers involves computing with numbers, and more often than not with fractional numbers rather than with integers. The general program-design rules that we have presented throughout this book apply to numeric calculations. There are also some additional considerations that apply specifically to numeric calculations. These considerations arise because of a very important but perhaps not obvious principle. To illustrate the principle, consider the following very simple piece of code, and predict its output:

```
X := 1/3 + 1/3 + 1/3;  
writeln(X)
```

The calculation hardly needs a computer. The expected output is 1. 0. Yet many computers will give an output such as

```
0. 9999
```

One need not be Sherlock Holmes to observe that the computer's performance is either incorrect or more subtle than our simple mental model of arithmetic. As it turns out, the second alternative is the better explanation. The numbers inside a computer are unlike the numbers you learned about in mathematics classes from grade school through calculus. Consequently, you must learn to think quite differently when performing involved numeric calculations on a computer. A detailed treatment of numeric programming techniques is beyond the scope of this book, but in this chapter we will describe some of the basic principles involved.

## A Hypothetical Decimal Computer

Most computers work in binary notation, and some of the problems that arise when one is doing numeric calculations are due to the differences between binary and decimal notation. However, the difference between the two ways of writing numerals is small, and the problems caused by this difference are typically small. We will discuss numeric calculations in terms of a fictitious computer that works in decimal (base ten) rather than in binary (base two) notation. Since we normally think in base ten, this will make the entire process easier to understand. Aside from the fact that it works in base ten, our hypothetical computer handles numbers in a very typical way.

In Chapter 1 we observed that most computers have their main memory divided into a series of locations called *words*. Numeric values are usually stored one value per word. The size of a word will vary from machine to machine, but usually all words in any one machine are of the same size. One word of our hypothetical computer has room for eight symbols, each either a sign or a decimal digit. Hence, a word may be diagrammed as follows:

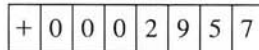
*word  
size*



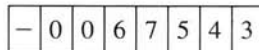
Each of the small boxes within a word can hold any one of the following 12 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -.

In our hypothetical computer, values of type *integer* are stored as their usual base ten numeral preceded by a sign. For example, the number 2957 would be stored as

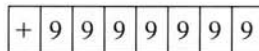
*storing  
integers*



The number -67543 would be stored as



The largest integer that can be stored in our computer is thus



which is one less than ten million. With Pascal implemented on our computer in this way, the value of *maxint* is 9999999. Similarly, the smallest possible negative integer value is minus this amount.

*maxint*

On our computer, values of type *real* are stored in what is called *floating point notation*. This is a variation on the E notation used to write *real* constants in Pascal. The computer word is divided into two parts. On our hypothetical computer, one part consists of five boxes and the other part consists of three boxes. The value of each real number is first converted to a form consisting of a decimal fraction multiplied by a

*floating point  
(real)  
numbers*

power of ten. For example, the value 123.4 would be converted to the equivalent form

$$+0.1234 \times 10^3$$

*fractional  
part*

The number with the decimal point in it is called the *fractional part* (or *mantissa*). The fraction part (including the sign) is stored in the first five boxes, and the exponent of 10 (including its sign) is stored in the last three boxes. So 123.4 is stored as

+	1	2	3	4	+	0	3
---	---	---	---	---	---	---	---

*exponent  
part*

The position of the decimal point is assumed to be before the first digit. It is not marked in any way in the computer word. The division between the exponent part and the fractional part is also fixed and understood by the computer. It is not marked in any way. (In our hypothetical computer, the boundary can be inferred by the presence of a plus or minus sign. However, in a typical computer, plus and minus would be represented by two digits, such as 0 and 1.)

*significant  
digits*

Other examples of storing values of type `real` are given in Figure 15.1. Notice that all numbers are normalized so that the fractional part has the decimal point immediately in front of the first nonzero digit. This is an attempt to preserve the maximum number of significant digits. Consider the number 0.01234. If the computer merely stores the first four digits after the decimal point, then the final digit 4 would be lost. However, because the computer normalizes the position of the decimal point, this number is stored as follows:

$$+0.1234 \times 10^{-1}$$

The normalization has saved that last digit. This moving of the decimal point is the origin of the term “floating point.”

*equality  
of reals*

There is room for only four decimal digits in our computer. For this reason, the value stored is sometimes only an approximation of the value we might expect a Pascal expression to represent. For example, 0.1234123 and 0.12340 have the same representation in our computer. Hence, on our hypothetical computer, the following boolean expression evaluates to `true`:

$$0.1234123 = 0.12340$$

As this example indicates, testing for equality between two values of type `real` is pointless and even dangerous.

Our computer rounds numbers when they have too many digits to fit in a word. For example, 765.46 is rounded to 765.5. Some systems truncate the number (i.e., discard the extra digits) instead. If our computer were to truncate instead of round, then the number 765.46 would be stored as 765.4 instead of 765.5.

*largest  
real  
number*

On our computer, the largest positive value of type `real` that we can store would look as follows in memory:

+	9	9	9	9	+	9	9
---	---	---	---	---	---	---	---



-0.1234E+03

-	1	2	3	4	+	0	3
---	---	---	---	---	---	---	---

0.01234

+	1	2	3	4	-	0	1
---	---	---	---	---	---	---	---

-0.001234

-	1	2	3	4	-	0	2
---	---	---	---	---	---	---	---

0.1234123

+	1	2	3	4	+	0	0
---	---	---	---	---	---	---	---

**Figure 15.1**  
Storing real  
(floating point)  
values.

Expressed more conventionally, this is the value

$$0.9999 \times 10^{99}$$

Hence, the largest possible value of type *real* is about  $10^{99}$ . By contrast, the largest possible value of type *integer* is only 9999999, or about  $10^7$ .

If a program attempts to compute a value of type *real* whose exponent is too large to fit into the space allocated for one exponent, that is called *real overflow* or *floating point overflow*. Similarly, if a program attempts to compute a value of type *integer* that is larger than `maxint` or smaller than the smallest integer that can be held in one word, that is called *integer overflow*. (Remember, small negative numbers are large in absolute value. For example,  $-9999999 < -1$ . Hence, an equivalent way to describe overflow is to say that it occurs when the computer attempts to produce numbers that are too large in magnitude, that is, too large in absolute value.) Whenever any sort of overflow occurs, an error message should be produced. However, many systems give no such error messages. They simply produce some meaningless value and keep on computing. At that point, the entire computation becomes meaningless.

*overflow*

In addition to there being a largest magnitude that can be stored as a value of type *real*, there is also a smallest possible magnitude. The smallest positive number of type *real* that can be stored in our machine will be produced by the following word configuration:

*smallest  
fraction*

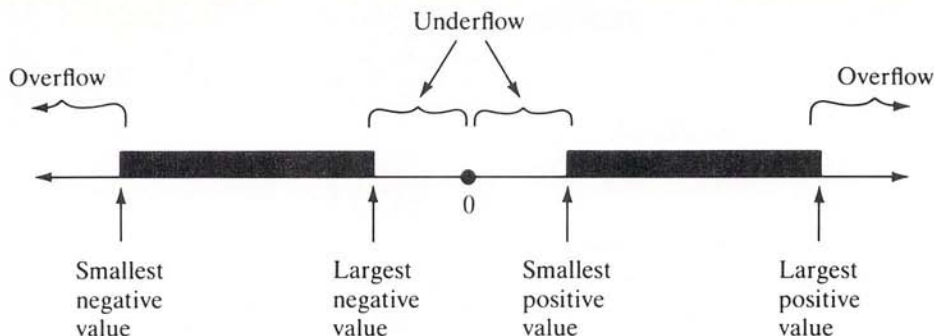
+	1	0	0	0	-	9	9
---	---	---	---	---	---	---	---

This is the number

$$0.1000 \times 10^{-99}$$

which is a decimal point followed by ninety nine zeros and then a one—certainly a very small number. However, calculations involving such small quantities do occur.

*underflow*



**Figure 15.2**  
Range of available  
real values.

When the computer attempts to produce a nonzero number whose absolute value is smaller than this quantity, that is called *real underflow* or *floating point underflow*. On many systems, any such small quantity is simply replaced by zero, which usually is a satisfactory approximation. Some systems give an error message in the case of floating point underflow. Unfortunately, a few other systems produce a meaningless result and continue the computation. The situation is similar to that of overflow but generally produces fewer problems. A diagram showing the ranges of overflow and underflow is given in Figure 15.2.

As you can see from the preceding discussion, the types `integer` and `real` are implemented in different ways. As an illustration, consider the difference between the constants `123412` and `123412.0`. The first is of type `integer` and is stored exactly as

+	0	1	2	3	4	1	2
---	---	---	---	---	---	---	---

The second is of type `real` and so is stored only as the approximate value

+	1	2	3	4	+	0	6
---	---	---	---	---	---	---	---

Most problems peculiar to numeric calculations arise because of the approximate nature of `real` values. When studying mathematics, we frequently think in terms of an ideal world where quantities are represented as exact values called “real numbers.” These exact numbers are not available on a computer. If the quantity we need requires more than four decimal digits after the decimal point in order to write it down, then it will not be represented exactly in our computer. If it consists of a decimal point followed by five or six nonzero digits, then we could use a computer with a larger word size. However, some quantities, such as the number  $\pi$  used in geometry, cannot be represented exactly by any finite string of digits. For these “real numbers” even a computer with a word size of one million digits could store only an approximation of the quantities they represent. Many of the “real numbers” of classical mathematics are simply not available on computers. Moreover, the missing numbers are not always very exotic ones. Recall the example that opened this chapter. It performs the calculation

$$X := 1/3 + 1/3 + 1/3$$

The number one-third has no representation as a finite string of decimal digits. Our computer will represent  $1/3$  as

$$0.3333 \times 10^0 = 0.33330$$

When three of these are added together, the result is 0.99990, rather than the value 1.0 that is predicted by the usual idealized model of arithmetic.

## Binary Numerals (Optional)

Most computers represent numbers in *binary notation* rather than in the more familiar base ten notation. On occasion, this can have a significant effect on the outcome of a numeric calculation. The basic idea of binary notation is quite simple. It is just like the base ten notation we normally use, except that the role of ten is replaced by the number two. Base ten notation uses the ten digits 0 through 9. Base two notation uses only two digits, 0 and 1. In base ten each change in position, from the rightmost to the leftmost digit, represents multiplying by ten. In base two each change in position, from the rightmost to the leftmost digit, represents multiplying by two.

For example, consider the ordinary base ten numeral 3019. It satisfies the following equality:

$$\begin{aligned} 3019 &= 3 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 9 \times 10^0 \\ &= 3 \times 1000 + 0 \times 100 + 1 \times 10 + 9 \times 1 = 3000 + 0 + 10 + 9 \end{aligned}$$

The meaning of any base ten numeral is decomposed in a similar way.

Next, consider an example of a binary (base two) numeral, such as 100101. The situation is the same except that now each digit position represents some power of two. For example,

$$\begin{aligned} &(\text{the base two numeral}) 100101 = \\ &(\text{the base ten expression}) 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\ &= 32 + 4 + 1 = 37 \text{ (base ten)} \end{aligned}$$

In binary notation the rightmost digit represents that digit multiplied by  $2^0 = 1$ , the next digit to the left represents that digit multiplied by  $2^1 = 2$ , the next multiplied by  $2^2 = 4$ , the next multiplied by  $2^3 = 8$ , and so forth. Any integer can be represented in this binary notation.

The treatment of fractions in binary notation is similar to that of decimal fractions. In decimal fractions, the digit positions after the decimal point represent smaller and smaller fractions. Each shift to the right represents dividing by ten. For example, in base ten,

$$\begin{aligned} 0.103 &= 1/10 + 0/10^2 + 3/10^3 \\ &= 1/10 + 0/100 + 3/1000 \end{aligned}$$

Fractions in binary notation follow the same principle, but with 10 replaced by 2. For example,

*example*

*whole  
numbers*

*fractions*



$$\begin{aligned}
 & \text{(the base two numeral) } 0.1101 \\
 & = \text{(the base ten expression) } 1/2 + 1/2^2 + 0/2^3 + 1/2^4 \\
 & = 1/2 + 1/4 + 0/8 + 1/16 = 0.8125 \text{ (base ten)}
 \end{aligned}$$

In binary notation the “point” is called a *binary point*, rather than a decimal point. The first digit after the binary point represents that number divided by  $2^1 = 2$ , the next digit after the binary point represents that digit divided by  $2^2 = 4$ , the next digit represents the digit divided by  $2^3 = 8$ , and so forth.

In both decimal and binary notation, any quantity between zero and one can be represented by a point followed by a string of digits. In both binary and decimal notation, this may sometimes require an infinite string of digits. For example, in base ten,

$$1/3 = 0.3333333333333333 \dots$$

As we add more 3s, we get a better approximation to  $1/3$ , but no finite number of digits after the decimal point will yield a number exactly equal to  $1/3$ .

A similar phenomenon occurs in binary notation. In binary notation, the fraction  $1/4$ , for example, can be expressed as the finite string 0.01, but the exact representation of  $1/5$  requires an infinite string of binary digits after the binary point.

$$1/5 = \text{(in binary) } 0.00110011001100110011 \dots$$

Any finite string of binary digits can only approximate the value  $1/5$ . In decimal notation  $1/5$  can be represented exactly as 0.2. As this example indicates, some quantities that we express exactly in decimal notation can become approximate quantities when stored in a binary computer.

In both binary and decimal notation, you can combine the notation for whole numbers and that for fractions. In base ten, 12.34 means 12 plus 0.34. In base two, 10.101 means 10 (base two) plus 0.101 (base two).

*binary  
arithmetic*

Arithmetic on binary numerals is very similar to arithmetic on base ten numerals. In particular, shifting the binary point is similar to shifting the decimal point in base ten. In base ten, shifting the point one position to the right is the same as multiplying by ten. In base two it is the same as multiplying by two. For example, in base two, 1.011 multiplied by two is 10.11, and 1.011 multiplied by four is 101.1.

---

## Machine Representation of Numbers in Binary (Optional)

*bits*

Like our hypothetical decimal computer, most real computers have a memory that is divided into locations called “words.” However, these words usually store strings of zeros and ones rather than strings of decimal digits. Recall that a digit that must always be either zero or one is called a *bit*. Computer word sizes are usually described as being some number of bits. Some typical word sizes are sixteen, thirty-two, and sixty-four bits. When people refer to a “sixteen-bit machine,” they mean that each word of the machine holds sixteen binary digits. (They do not mean that the computer costs \$2.)

Since computer words usually hold bits, most computers store numbers in binary notation. Aside from the fact that the numbers are expressed in binary notation, the

---

method of representing numbers is the same as we described for our hypothetical decimal machine.

Numbers of type `integer` are stored as binary numerals, either in the exact form described in the previous section or in some variant of that notation. Each value of type `integer` is stored in one word. For example, the number five has the binary representation 101 and, in a sixteen-bit word, it might be stored as

*integers*

+	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A word holds only zeros and ones. What we have written as the sign + must therefore be represented as a zero or one. If we take 0 to stand for plus and 1 to stand for minus, then in this sixteen-bit computer, the preceding word's contents would really be

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

We will always use the plus and minus sign rather than 0 and 1 to denote the sign of a number in storage. It helps avoid confusion.

Since one bit is occupied by the sign, the largest integer that can be stored in a sixteen-bit computer is the number with binary representation consisting of fifteen 1's. In base ten that number is written 32767. You can compute this base ten numeral by evaluating the sum

*largest  
integer*

$$1 \times 2^{14} + 1 \times 2^{13} + 1 \times 2^{12} + 1 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

(If you understand a bit of binary arithmetic, you can calculate it more quickly as follows:

$$1111111111111111 \text{ (base two)} = 1000000000000000 - 1 \text{ (base two)} = \\ \text{(in base ten)} 2^{15} - 1 = 32768 - 1 = 32767$$

However, if you are uncomfortable with binary arithmetic, simply do it the long way.)

Thus, if you are working on a sixteen-bit machine, you can expect the value of `maxint` to be 32767. The smallest negative number your sixteen-bit machine can hold will probably be about -32767. This is not part of the definition of the Pascal language. The exact way that numbers are represented is left up to the implementers. Hence, the value of `maxint` and the value of the smallest negative integer in your machine might vary somewhat from these figures. However, these values will be approximately correct for most sixteen-bit machines.

Aside from the fact that binary notation is used, numbers of type `real` are stored in the same way as we described for our decimal computer. For example, a sixteen-bit word might be divided to allow four digits to express the exponent and twelve digits to express the fractional part. On a binary machine, the exponent represents a power of two rather than a power of ten. For example, consider the following word configuration for such a machine:

*real  
numbers*

+	1	0	1	1	0	0	0	0	0	0	0	0	+	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

It represents the number

$$(\text{in binary}) 0.1011 \times 2^{101} = (\text{in decimal}) (1/2 + 0/4 + 1/8 + 1/16) \times (2^5)$$

We have taken the liberty of using the digit 2 in our binary expression. This mixed notation is sometimes easier to understand than absolutely pure binary notation.

Although numbers are invariably stored in binary notation, that notation normally has little, if any, effect on the outcome of a numeric computation. Therefore, we will end our discussion of binary arithmetic here and return to using our hypothetical decimal computer.

---

## Extra Precision (Optional)

Some computer installations have facilities to store numbers in more than one word and so obtain a more accurate representation for values of type `real`. This is not part of the definition of Pascal and is not available on most Pascal systems. However, it is a common feature of other programming languages and does occur in some Pascal implementations.

*double  
precision*

One common method for obtaining extra accuracy when storing `real` values is called *double precision*. In double precision, each `real` number is stored in two words. This yields more than double the number of meaningful digits, because all the extra digits of the second word normally go into the fractional part. Our hypothetical decimal computer had an eight-decimal-digit word size. On that computer, a typical double precision implementation would store one `real` value in two words, as illustrated by the example in Figure 15.3. In our hypothetical decimal computer, this accurately represents the decimal number

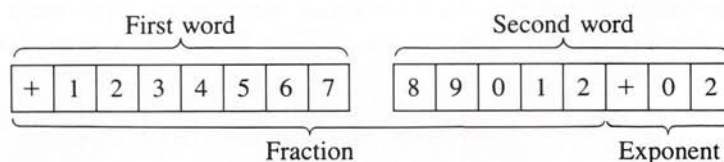
12.3456789012

With this sort of double precision, the largest possible `real` value that can be stored is about the same as it is for the normal one-word representation. However, a full twelve decimal digits of accuracy can be represented. The ordinary one-word representation, allowed for only four decimal digits of accuracy.

The disadvantages of double precision are that it uses more storage and that it usually causes programs to run more slowly.

There is no standard Pascal syntax for double precision numbers. You will have to consult the documentation for your particular system to see if it is available and, if available, to see how to use it in a Pascal program.

**Figure 15.3**  
(Optional)  
Example of a  
double-precision  
number.





## Self-Test Exercises

1. Describe how each of the following integer and real constants are represented in our hypothetical decimal computer:

123456	-123456
123.456	-123.456
0.00123123	-0.00123123
3.14159265358979323846	

2. Our hypothetical decimal computer had a word size of eight. To store values of type `real`, it used five digit positions for the fractional part and three digit positions for the exponent part. Suppose that we instead used other combinations. At what value would `real` overflow occur if we instead used the following combinations?

- Fractional part uses four digit positions and exponent part uses four.
- Fractional part uses three digit positions and exponent part uses five.
- Fractional part uses six digit positions and exponent part uses two.

3. (This exercise applies to the optional section “Binary Numerals.”) Convert the following binary numerals into equivalent decimal numerals: 111, 101, 100, 11011, 010110, 0.1, 0.01, 0.001, 0.101, 1.001, 101.101

4. (This exercise applies to the optional section “Machine Representation of Numbers in Binary.”) What would you expect as the value of `maxint` in a thirty-two-bit machine? Assume that numbers are stored as described in this chapter. What about a sixty-four-bit machine?

---

“So so” is good, very good,  
very excellent good; and yet it is not;  
it is but so so.

*William Shakespeare, As You Like It*

---

## Pitfall

### Sources of Error in real Arithmetic

In computations with values of type `real`, errors arise because numbers are stored as approximate values. These approximations are sometimes accurate enough and other times very inaccurate. In this and the next section we will discuss some common sources of inaccuracy in programs that compute values of type `real`.

As you will recall, overflow results when the computer tries to compute

*overflow*

*underflow*

a number larger than it can hold in memory. The problem of integer overflow can sometimes be avoided by using variables of type `real` to do calculations involving large numbers, even if the quantities involved are whole numbers. The computer can store much larger values of type `real` than it can values of type `integer`. There is a certain loss of accuracy in doing this, but often this loss of accuracy is tolerable.

Recall that real underflow occurs when the computer attempts to produce a nonzero value of type `real` that is too small in absolute value, that is, too close to zero. These values cannot be represented in memory. Most computers, including our hypothetical decimal computer, simply estimate such values as zero and store a zero as the result of the calculation. This is called *rounding to zero*. If your computer rounds to zero on underflow, then underflow will seldom be a problem. If your computer does anything else, then you must be careful to avoid underflow.

*multiplication  
and  
division*

When a multiplication or division is performed, the answer usually has more digits than either of the two numbers being combined. Frequently, these extra digits cannot be represented in memory and so are lost, along with a little bit of accuracy. For example, consider the following code: (Here and in the examples that follow, we will set the values of variables by means of assignment statements. This is to keep the examples small. In practice, these values might be read from the keyboard or might be the results of other calculations.)

```
X := 912.0;
Y := 0.11;
Z := X * Y;
```

The value that should be stored in Z is

$$912.0 \times 0.11 = 100.32$$

However, our hypothetical decimal computer only stores four digits in the fractional part of a `real` value. Hence, it will store the value of Z as

+	1	0	0	3	+	0	3
---	---	---	---	---	---	---	---

which represents the value

$$0.1003 \times 10^3$$

This means that the last digit is lost, and the value of Z becomes 100.30. Even if the values of X and Y were completely accurate, the value of Z has lost one digit of accuracy as the result of a simple multiplication. In all the examples of this section and the next, we will use our hypothetical decimal computer. Hence, we are only allowed four digits of accuracy.

*simple  
addition*

Even very simple addition or subtraction can produce a slightly inaccurate result. If the computer adds the number 9.222 to itself, the result should be

$$9.222 + 9.222 = 18.444$$

However, our computer only retains four digits and so will store the answer as

$$0.1844 \times 10^2$$

This means that 18.444 was rounded to 18.440, and one digit of the answer was lost.

Under some circumstances, the loss of accuracy in addition can be dramatic. Consider the following piece of code:

```
X := 2000.0;
Y := 0.4;
X := X + Y
```

The values of X and Y are stored as

```
0.2000 × 104
0.4000 × 100
```

Like most computers, our hypothetical decimal machine cannot add two numbers unless they have the same exponent part. Hence, it must change one of the two numbers. On our machine the second number is changed to

```
0.00004 × 104
```

Then the following addition is performed

```
0.2000 × 104
+0.00004 × 104
-----
0.20004 × 104
```

This answer is what we might expect as the value of X, but unfortunately that is not the value stored. Since our computer stores only four digits after the decimal point, it stores the following as the value of X:

```
0.2000 × 104
```

The adding in of Y thus had absolutely no effect on the value of X.

Situations like the one in the preceding example are common. To avoid this sort of problem, you must somehow avoid adding or subtracting two values of very different size. Sometimes this can be done by rearranging the order in which numbers are combined. As an example, consider the following code:

```
X := 2000.0;
Y := 0.4; U := 0.3; V := 0.4;
X := X + Y;
X := X + U;
X := X + V
```

As we have just seen, adding Y has no effect on the value of X. Similarly, adding U and adding V each have no effect. This calculation leaves the value of X unchanged at:

```
0.2000 × 104
```

However, if we first combine Y, U, and V to obtain a larger number and then combine this larger number with X, the resultant value of X will be close to the value we expect.

*adding  
large and  
small  
numbers*



Consider the following slightly different code for the same computation:

```
X := 2000.0;
Y := 0.4; U := 0.3; V := 0.4;
W := Y + U + V;
X := X + W
```

The value of W is obtained by first adding the values of Y and U and then combining that sum with the value of V:

$$\begin{array}{r} 0.4000 \times 10^0 \\ + 0.3000 \times 10^0 \\ \hline 0.7000 \times 10^0 \\ \\ 0.7000 \times 10^0 \\ + 0.4000 \times 10^0 \\ \hline 1.1000 \times 10^0 = 0.1100 \times 10^1 \end{array}$$

This value is then added to the value of X, as follows:

$$\begin{array}{r} 0.2000 \times 10^4 \\ + 0.00011 \times 10^4 \\ \hline 0.20011 \times 10^4 \end{array}$$

The final value of X is stored as

$$0.2001 \times 10^4$$

By rearranging the order of the additions, we have added one digit of accuracy to the answer. This is a standard trick. If the small numbers are first added together, then that will produce a somewhat larger value. This larger value can then be combined with other large values. In this way the numbers being combined are more nearly equal, and so the results of their addition will be more accurate.

## Pitfall

### Error Propagation

Each individual operation on a value of type `real` is likely to introduce only a very small error. However, after a number of operations, these small errors may be compounded to produce a very large inaccuracy. Again, we illustrate the pitfall with a piece of code:

```
. . .
B := 0.1232;
C := A - B;
D := 10000.0;
X := C * D
```

The three dots represent some computation that sets the value of A. Let us say that A gets set to 0.1234, a value very close to that of B. The value of C gets set by the calculation

$$\begin{array}{r} 0.1234 \times 10^0 \\ -0.1232 \times 10^0 \\ \hline 0.0002 \times 10^0 = 0.2000 \times 10^{-3} \end{array}$$

The value of X is then computed by multiplying that value by 10,000 to obtain the following value of X:

$$0.2000 \times 10^1$$

So far things look fine. The answer appears to be 2. However, as we have already seen, it is easy for a program to calculate a value that is slightly in error. Suppose that the value of A was slightly in error. Specifically, suppose the correct value of A is 0.1233, rather than 0.1234. Then the correct answer is

$$(0.1233 - 0.1232) \times 10,000 = 1$$

The correct answer is 1, but our code computed it as 2. A slight mistake has been compounded, and our answer is now wrong by a factor of two.

The problem is that when subtraction is performed on two nearly equal numbers, the answer is the difference between the end digits of the two numbers. After the subtraction, only the last digits of the two almost-equal numbers have any effect on the rest of the computation. But these are exactly the digits that are likely to be incorrect. Hence, a program should somehow avoid subtracting two almost-equal numbers of type `real`.

All the examples in this and the previous section are unrealistic in the sense that most computers represent `real` values with an accuracy equivalent to more than four decimal digits. In all other respects, they are real pitfalls. Realistic examples can be manufactured simply by adding a few more digits to the initial values and leaving the rest of the code unchanged.

*subtracting  
almost equal  
numbers*

---

## Case Study

---

### Series Evaluation

#### Problem Definition

One common numeric task is to sum a series. For example, consider the following series:

$$\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{N}{2^N}$$

---

Suppose that we have already declared a function called `Power` such that the value of `Power(x,y)` is

$$x^y$$

Our task is to compute the value of the sum given the value of  $N$ .

### Discussion

For concreteness, suppose that the value of  $N$  is 100. The most obvious way to calculate the sum is as follows:

```
Sum := 0;
for I := 1 to 100 do
    Sum := Sum + (I/Power(2, I))
```

If the calculation is carried out with complete accuracy, this will set the value of `Sum` to the desired value. However, the operations are not carried out with complete accuracy. Moreover, the values of the successive terms rapidly become very small, while in comparison, the value of `Sum` remains moderately large. Hence, after the first few iterations, the loop is adding two numbers of very different size. As we have seen, this can lead to inaccuracies in the answer.

We can avoid adding numbers of such greatly differing size by summing the series in the other direction, like so:

```
Sum := 0;
for I := 100 downto 1 do
    Sum := Sum + (I/Power(2, I))
```

The numbers being combined will then be more nearly equal, and hence the results of the additions are likely to be more accurate.

The general algorithm is the same but with 100 replaced by  $N$ .

```
Sum := 0;
for I := N downto 1 do
    Sum := Sum + (I/Power(2, I))
```

*order of  
summation*

**ALGORITHM**

---

## Case Study

---

### Finding a Root of a Function

A common numeric programming task is to solve an equation. For example, consider the equation

$$x^3 + 2x = 33$$


---



One solution is 3, since,

$$3^3 + 2 \times 3 = 33$$

By rearranging the equation, we can always get it into the form

$$F(x) = 0$$

where  $F(x)$  is some expression that can be made into a Pascal function having one argument of type `real` and returning a value of type `real`. For example, the equation

$$x^3 + 2x = 33$$

can be rearranged to the following equivalent equation:

$$x^3 + 2x - 33 = 0$$

The expression on the left-hand side is computed by the Pascal function declared as

```
function F(X: real): real;
begin
  F := X * X * X + 2 * X - 33
end;
```

Solving equations of this form is equivalent to finding a value  $x$  such that the expression  $F(x)$  on the left side of the rearranged equation is made equal to zero. Such a value  $x$  is called a *root* of the function  $F(x)$ .

*roots*

## Problem Definition

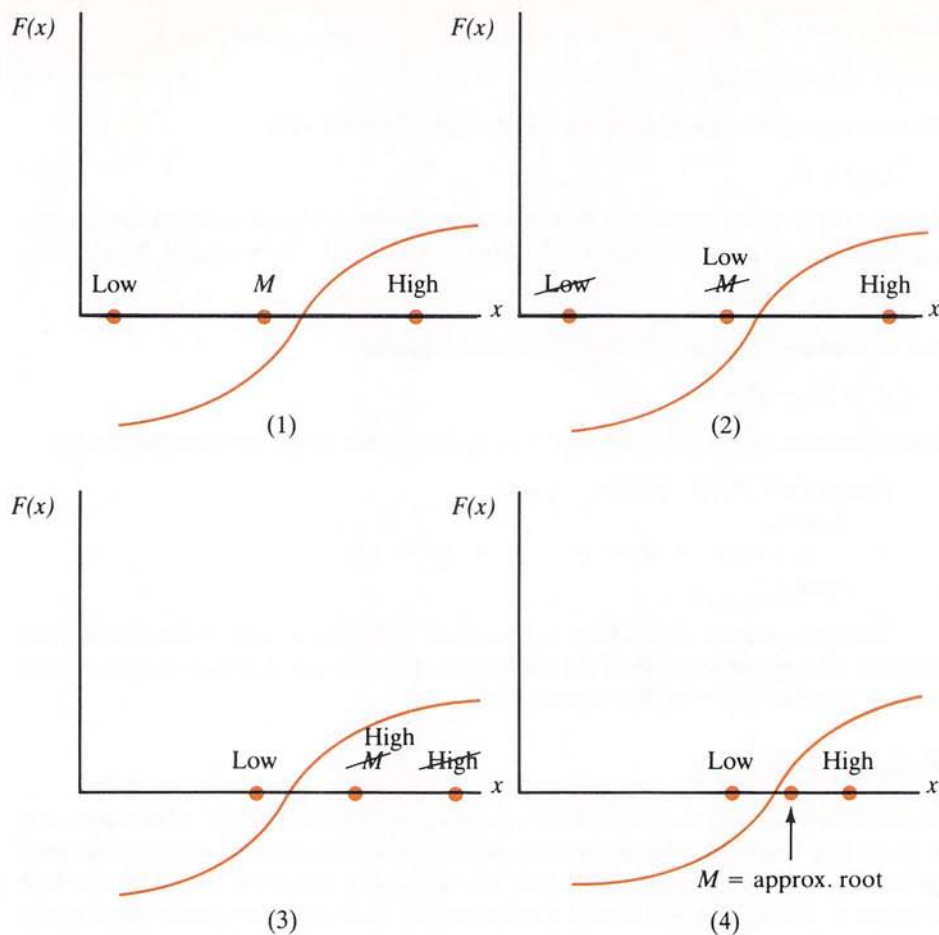
In this section we will design a program to find a root of a function  $F$ . The function  $F$  is given as a Pascal function declaration, which we will incorporate into our final program. The user will provide two values  $x_1$  and  $x_2$  such that there is exactly one root between  $x_1$  and  $x_2$ . We will design a program that finds the approximate value of the root of the function  $F$ . The method works for a wide range of different functions  $F$ . We need only assume that the graph of the function can be drawn on paper as a smooth line. (The technically precise condition is that the function must be *continuous*. However, we will not stop to define that term. The informal notion of “easy to draw as a smooth line” will do here.)

## Discussion

Our goal is to find a value  $M$  such that  $F(M)$  is approximately equal to zero. The method we will use is called the *bisection method*. (The technique is similar to that of the binary search algorithm we discussed in Chapter 14, but you need not have read that section in order to understand this method.) The idea of the bisection method is depicted in Figure 15.4. The graph represents the function  $F$ . Two values, *Low* and *High*, are chosen so that exactly one root lies between them. We therefore know that the following relation holds:

$$\text{Low} < \text{root} < \text{High}$$

*bisection  
method*



**Figure 15.4**  
One way to find a  
root.

The midpoint between these two values  $Low$  and  $High$  is then computed. In Figure 15.4, this midpoint is denoted by  $M$ . The root is either between  $Low$  and  $M$  or else it is between  $M$  and  $High$ . For now, assume that we can tell which of these two intervals contains the root; later we will return and figure out a way to do so. In the figure the root is between  $M$  and  $High$ . This process has managed to narrow down the location of the root. Originally we knew it was between  $Low$  and  $High$ . Now we know that it is in the smaller interval between  $M$  and  $High$  (or, in other cases, between  $Low$  and  $M$ ). Next, we change the values  $Low$  and  $High$  to these new endpoint values; in the figure example,  $M$  becomes the new value of  $Low$ , and the value of  $High$  is unchanged. If we keep repeating this process, we eventually get a very small interval that contains the root. This gives us an approximation to the value of the root. The method is outlined in the following pseudocode:

request values  $x_1$  and  $x_2$  such that

$$x_1 < \text{root} < x_2;$$

Low :=  $x_1$ ; High :=  $x_2$ ;

M := (Low + High) / 2;

RootFound := false;

while not (RootFound) do

begin

if (F(M) is approximately equal to 0.0) then

RootFound := true

else if the root is between Low and M then

High := M

else if the root is between M and High then

Low := M;

M := (Low + High) / 2

end;

output: "The root is approximately M. "

ALGORITHM

We still must design a subalgorithm to decide whether the root is between Low and M or between M and High. The interval containing the root can be determined from the signs of the values  $F(\text{Low})$ ,  $F(M)$ , and  $F(\text{High})$ . One of the two values  $F(\text{Low})$  and  $F(\text{High})$  will be positive and one will be negative. If the sign of  $F(M)$  matches that of  $F(\text{High})$ , then High gets its value changed to M. If the match is with Low, then Low gets changed. So the pseudocode for the nested *if-then-else* statement can be refined to the following:

ALGORITHM  
refinement

if (F(M) is approximately equal to 0.0) then

RootFound := true

else if SameSign(F(High), F(M)) then

High := M

else if SameSign(F(Low), F(M)) then

Low := M

where SameSign is a boolean-valued function that tests two values to see whether they have the same sign.

The test for approximate zero will depend on a constant called Threshold. As long as a number is less than Threshold in absolute value, it will be considered close enough to zero. The value of Threshold will depend on the accuracy of the computer and the accuracy needed for the particular application. Hence, we will ask the user to supply the value. The test for approximate zero can then be expressed as

$$\text{abs}(F(M)) \leq \text{Threshold}$$

The final program is given in Figure 15.5. The function declaration for F is indicated by three dots. It can be filled in with any function definition that satisfies the assumptions we have made.

If incorrect initial values are used, the algorithm expressed in our pseudocode can go into an infinite loop. Hence, in the final program, we have placed a limit on the

additional  
input needed

additional  
error checks



```

program FindRoot(input, output);
{Finds a root of the function F by the bisection method.}
const MaxIterations = 1000;
var Low, High, M: real;
    Threshold: real;
    RootFound: boolean;
    Count: 0 .. MaxIterations;

function F(X: real): real;
{The function whose root is being sought.}
    . . .

function SameSign(V1, V2: real): boolean;
{Returns true if V1 and V2 have the same sign; otherwise returns false.}
begin {SameSign}
    if (V1 >= 0.0) and (V2 >= 0.0) then
        SameSign := true
    else if (V1 <= 0.0) and (V2 <= 0.0) then
        SameSign := true
    else
        SameSign := false
end; {SameSign}

procedure ReadInterval(var Low, High: real);
{Reads in two values that are supposed to have exactly one root
between them. Makes a test to see if they are plausible values.}
begin {ReadInterval}
    repeat
        writeln('Enter two values,');
        writeln('the first less than the root,');
        writeln('the second greater than the root. ');
        writeln('Be sure there is');
        writeln('exactly one root between them. ');
        readln(Low, High);
        if SameSign(F(Low), F(High)) then
            begin{then}
                writeln('Those cannot be right. ');
                writeln('Try again. ')
            end{then}
        until not (SameSign(F(Low), F(High)))
    end; {ReadInterval}

```

**Figure 15.5**  
**Program to find a**  
**root of a function.**

```

begin {Program}
  writeln('Enter accuracy desired. ');
  readln(Threshold);
  RootFound := false;
  ReadInterval(Low, High);
  M := (Low + High)/2;
  Count := 0;

  while not(RootFound) and (Count < MaxIterations) do
    begin {while}
      {Low < (the root) < High and M = (Low + High)/2}
      if abs(F(M)) <= Threshold then
        RootFound := true
      else if SameSign(F(High), F(M)) then
        High := M
      else if SameSign(F(Low), F(M)) then
        Low := M
      else
        writeln('Something is wrong! ');

      M := (Low + High)/2;
      Count := Count + 1
    end; {while}

    if Count = MaxIterations then
      writeln('Exceeded iteration limit.')
    else
      writeln('The root is approximately', M)
  end. {Program}

```

**Figure 15.5**  
(continued)

number of loop iterations allowed. When that limit is exceeded, the program halts and reports that something is likely to be amiss. As an additional check on faulty data, we have added a clause with an error message at the end of the nested *if-then-else* statement within the main program loop. If the data and function are as they should be, then this clause will not be executed. However, incorrect data could cause all of the boolean expressions to be false. The extra clause will catch this situation immediately.

---

As a wise programmer once said, “Floating point numbers are like sandpiles: every time you move one, you lose a little sand and you pick up a little dirt.” And after a few computations, things can get pretty dirty.

*B.W. Kernighan and P.J. Plauger,  
The Elements of Programming Style*

---

## Summary of Problem Solving and Programming Techniques

- The largest value of type `real` that an installation can accommodate is always much larger than the largest value of type `integer` that it can handle. Hence, one way to avoid `integer` overflow is to represent quantities as values of type `real` even though they are whole numbers.
- Values of type `real` are stored as approximate quantities. Hence, computations involving these numbers yield only approximations of the desired results. Unless particular care is taken to minimize errors, these approximations can often be very inaccurate.
- Since `real` values are stored as approximate quantities, any test of two `real` values for exact equality yields a meaningless result.
- Some common sources of error in programs involving the type `real` are round-off error in any arithmetic operation, such as addition or multiplication, but most especially in certain combinations such as adding two numbers of very different size or subtracting two numbers of almost equal size.

---

## Summary of Terms

### floating point numbers

In Pascal, numbers of type `real`.

### overflow

The condition that results when a program attempts to compute a numeric value whose magnitude is too large. More precisely, it is the condition that results when a program attempts to compute a numeric value that is larger than the largest value of that type that the computer can represent in memory, or is smaller than the smallest negative value of that type that the computer can represent in memory.

### underflow

The condition that results when a program attempts to compute a value of type `real` such that the value is smaller in absolute value than the smallest positive `real` value that the system can represent in memory. In other words, it is the condition that results when a program attempts to compute a nonzero value that is too close to zero to be represented in memory (except possibly by the approximately equal value of zero).

---

## Exercises

### Self-Test Exercises

5. Assume that Pascal arithmetic is implemented as we described for our hypothetical decimal computer. What is the output of the following when embedded in a complete program with the variables declared to be of type `real`?
-



```

X := 300.00;
Y := 0.12345678;
writeln(X + Y)

```

6. Assume that Pascal arithmetic is implemented as we described for our hypothetical decimal computer. What is the output of the following when embedded in a complete program with the variables declared to be of type `real`?

```

X := 1.000E-90;
Y := 1.000E-25;
Z := 1.000E+25;
writeln(X * Y * Z, X * Z * Y)

```

Remember that multiplication is performed in order from left to right and that our hypothetical computer rounds to zero on floating point underflow.

### Interactive Exercises

7. Find (approximately) the largest value for the constant `Epsilon` that will cause the following `writeln` to be executed on your system:

```

X := 1.0 + Epsilon;
if X = 1.0
  then writeln('It's really nothing')

```

8. Type up and run the program in Figure 15.5. Use the following declaration for `F`:

```

function F(X: real): real;
begin{F}
  F := X * X - 4
end; {F}

```

The exact root is thus 2.0. The program will approximate that root.

### Programming Exercises

9. Write a function declaration for a function called `Digit` that returns the value of the  $n$ th digit from the right of an integer argument. The value of  $n$  should be a second argument. For example, `Digit(9635, 1)` returns 5, and `Digit(9635, 3)` returns 6.

10. If  $e$  denotes the base of the natural logarithm then the value  $e^x$  can be calculated by the series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \dots$$

Write a program that computes an approximate value of  $e$  by summing that series for  $N$  terms. Have the program compute the series from left to right and from right to left, and then output both results. The value  $e^x$  can also be computed by the standard function `exp(x)`. Have the program also output the value of  $e$  calculated by `exp(1.0)`. Compare the three results. Embed these three calculations in a loop that repeats the

calculation for values of *N* from 1 to 100. To avoid integer overflow, store the factorials as values of type *real* (or avoid using them altogether).

11. Write a program to sum the following series from left to right until a term whose absolute value is less than 0.00001 is encountered, and to then output the answer:

$$4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \dots$$

(The denominators are the positive odd numbers 1, 3, 5, 7, 9, 11, . . .) Have the program then recalculate the sum from right to left, using the same number of terms and output that value as well. Compare the two results.

12. (This exercise applies to the optional section “Binary Numerals.”) Write a program that takes base two numerals (for whole numbers) as input and outputs the equivalent base ten numeral.

13. (This exercise applies to the optional section “Binary Numerals.”) Write a program that takes base ten numerals (for whole numbers) as input and outputs the equivalent base two numeral.

14. (This exercise applies to the optional section “Binary Numerals.”) A *hexadecimal numeral* is a numeral written in base sixteen. Write a program that takes a hexadecimal numeral (for a whole number) as input and outputs the equivalent base ten numeral. Use the first six letters of the alphabet for the digits “ten” through “fifteen.”

15. (This exercise applies to the optional section “Binary Numerals.”) Write a program that takes base ten numerals (for whole numbers) as input and outputs the equivalent hexadecimal numeral. (See the previous exercise for a definition of hexadecimal numerals.)

16. One way to obtain extra digits is to store numbers as arrays of digits. Write a program that reads in two whole numbers with up to 20 digits each and stores their digits in arrays of type

```
array[0 . . 19] of integer
```

The program then computes the sum of the two numbers, stores the result in an array of the same type, and outputs the result to the screen. Use the ordinary addition algorithm that you learned in grade school. Be sure to issue an “overflow” message if the result is more than 20 digits long.

17. It is wasteful to store just one digit in an array location that can hold about the number of digits in *maxint*. Redo the previous exercise, but this time store *L* digits in each array variable, where *L* is two less than the length of *maxint* written in base ten. You will need to modify the addition algorithm slightly, but the idea is still the same. Use 0 . . 4 as the array index type.

18. Do the previous exercise for multiplication instead of addition.

19. Use the ideas in the previous exercise to design a program that can perform multiplication of “real” numbers that yields at least twice as many significant digits as your system’s ordinary Pascal *real* multiplication does.

20. (This exercise assumes that you know what a derivative is.) If  $x$  is a maximum or minimum of a function  $f$ , and  $f$  has a derivative at  $x$ , then the derivative of  $f$  is zero at  $x$ . Use this idea to design a program to find local minima and maxima of polynomials of degree two. Use any input format that is convenient.
21. Redo the previous exercise, but allow polynomials of arbitrary degree, and use the bisection method to find the roots of the derivatives.

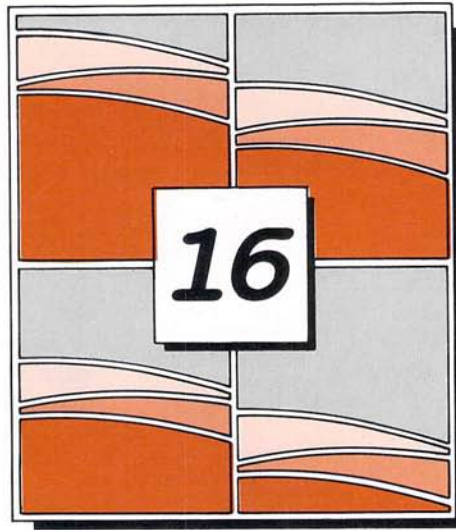
---

## References for Further Reading

- B.W. Kernighan and P.J. Plauger, *The Elements of Programming Style*, 1978, McGraw-Hill, New York. Includes material on pitfalls in both numeric and nonnumeric programming. The examples are in Fortran and PL/I, not in Pascal.
- D.E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, 2nd ed. 1981, Addison-Wesley, Reading, Mass. Also does not use Pascal, but the text can be read without reading the programs.
- T. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 1980, Springer-Verlag, New York. Uses Algol, a language similar to Pascal.
-







## ***More File Types***

A little more than kin, and less than kind.  
*William Shakespeare, Hamlet*

## Chapter Contents

The General Notion of a File  
File Variables  
TURBO Pascal—Opening Files  
Standard Pascal—Opening Files  
Windows  
read and write  
TURBO Pascal Case Study—  
Processing a File of Numeric Data  
Pitfall—Unexpected End of File  
Self-Test Exercises  
TURBO Pascal Example—Creating  
a File of Records

Deciding What Type of File to Use  
TURBO Pascal—seek  
TURBO Pascal Examples—Random  
and Sequential File Access  
Files as Parameters to Procedures  
TURBO Pascal Case Study—Merging  
Two Files  
Summary of Problem Solving and  
Programming Techniques  
Summary of Pascal Constructs  
Exercises

**T**ext files, which we have already used, are a special case of the more general construct known as “files.” In this chapter we describe Pascal files in complete generality and also present programming and problem solving techniques that use files of types other than `text`. As we pointed out in Chapter 13, file handling is highly implementation dependent. Files are treated slightly differently in standard Pascal and TURBO Pascal. As was true in Chapter 13, the emphasis in this chapter will be on TURBO Pascal, but we also include a brief description of file handling in standard Pascal. Those sections that are marked neither “Standard Pascal” nor “TURBO Pascal” apply equally to both types of systems. However, all the programming examples are given in TURBO Pascal. They will run without change on TURBO Pascal systems, but they would require some minor modifications before they would work on standard Pascal systems.



---

## The General Notion of a File

Files are used for holding data in secondary storage so that the data may remain after the program has run to completion. At some later time the same or another program may access the data in the file. A file is a named collection of data in secondary storage. In any kind of file, all the data must be of the same type. A text file is a special kind of file in which the data is all characters. In general, the data in a file may be of almost any Pascal data type. All types of files are similar in nature, but text files have some additional properties that are not shared by other file types. Files of type `text` can be read by using an editor and so are accessible by means other than a Pascal program. Other types of files *cannot normally be accessed by an editor; they can only be used as input and/or output data for Pascal programs*. This is because data is represented differently in text files and in nontext files. With text files, all data is converted into characters and so can be read by the editor. In nontext files, data is stored using the same binary encoding that is used in main memory to encode variable values of that data type. Since the data is not converted into characters, these files cannot be read using the editor.

In all versions of Pascal, text files must be accessed sequentially proceeding from the first character to the next without any backing up. In standard Pascal this restriction applies to nontext files as well. However, in TURBO Pascal nontext files are not restricted in this manner. TURBO Pascal allows data in nontext files to be accessed in any order. For this reason nontext files are called *random access files* in TURBO Pascal manuals. Other books refer to these files as *binary files*. We will simply refer to them as *nontext files*.

A *file* consists of a sequence of items called *components*, all of which are values of some one type known as the *component type*. The component type of a file can be almost any of the data types we have seen. The component type may be a simple type, such as `integer` or `char`. It might be an array type, such as an array of integers. It very frequently is a record type. The only data types that are not allowed are file types, and structured types that involve files, such as an array of files. In particular, you cannot have a file of files.

This description makes a file sound very much like an array, and indeed, a file is conceptually very much like a one-dimensional array. There are, however, two important conceptual differences between a file and an array. First, the size of an array must be declared in advance and so is bounded by some fixed number. On the other hand, there is no limit to the size of a file. The number of components that are placed in a file is not declared anywhere in the program, and there is no limit to the number of such components. Any particular implementation will impose an upper bound on the size of a file, but this is typically so large that, for most purposes, it can be considered unbounded. Second, files are kept in secondary storage and so can remain in storage after the program has run to completion. Hence, files are more permanent than arrays.

A type definition for a file type consists of the two words *file of* followed by the component type. For example, if you want `FileInt` to be the name of a type consisting of a file of integers, then the type declaration is as follows:

```
type FileInt = file of integer;
```

*random access  
files*

*binary files*

*component  
type*

*comparison  
to arrays*

*syntax  
for type  
definitions*

Below is a sample type declaration section that declares three file types:

```

type List = array[0 . . 10] of integer;
      Item = record
                Name: array[1 . . 20] of char;
                Number: integer;
                Price: real
            end;
FileType1 = file of real;
FileType2 = file of List;
FileType3 = file of Item;

```

*text  
files*

A text file is almost the same thing as a *file of char*. The only difference is that a text file is divided into lines and a *file of char* is not. Despite their similarities, text files and other types of files are usually thought of as two different categories of files. Moreover, text files and other files are treated slightly differently by the standard procedures for reading from and writing to files. Because of these small differences, it is best to treat text files as a special category of files; when compared to other types of files, text files are “more than kin and less than kind.” To avoid problems, use the descriptions presented in this chapter for files other than text files, and use the descriptions presented in Chapter 13 for text files.

The details for opening nontext files, declaring file variables, and even much of the use of `read` and `write` are the same as, or very similar to, what they are for text files, and so we can be brief in presenting the details.

---

## File Variables

Variables of file types are declared in the same way as other variables. For example,

```
var Y: file of integer;
```

An alternative, and usually preferable, way to declare `Y` would be to declare the type `FileInt` as in the previous section and to declare `Y` by

```
var Y: FileInt;
```

Within a Pascal program, file variable names are used to refer to files when retrieving input from the file or producing output to the file. The situation for other types of file variables is similar to that for file variables of type `text`.

## TURBO Pascal

---

### Opening Files

*assign*

The use of `assign` for other types of files is the same as what we described in Chapter 13 for text files. In a TURBO Pascal program, all files have both a string name and a file variable name. Before a TURBO Pascal program can do anything with a file, it must



associate a file variable name with the string name for the file. This association is accomplished with a call to the predefined procedure `assign`. The details are the same as those that we described for text files, except that the file variable type must be declared in the manner we described in the previous section. After a call to `assign`, the program refers to the file by its file variable name and not by its string name.

String names for nontext files, like string names for text files, have an optional extension of up to three letters. Since `.TXT` is used for text files, it should not be used for nontext files. The extension `.DAT` is often used for nontext files, since it suggests that the file contains “data.”

The syntax for using `reset` and `rewrite` with nontext files is identical to the syntax for using them for text files. However, the usage is slightly different. With text files `reset` is used to open files for reading, and `rewrite` is used to open them for writing. With nontext files, it is possible to mix reading from and writing to any nontext file which is opened with either procedure. With nontext files, `reset` is used to open *existing files*, and `rewrite` is used to create *new files*. In TURBO Pascal any nontext file that already exists in secondary storage is opened with the predefined procedure `reset`. A new file is created and opened by a call to the predefined procedure `rewrite`. If an existing file is opened with `rewrite`, then the contents of the file will be lost. `rewrite` always gives a blank file.

Once a file has been opened, a program can read from it using the procedure `read` and write to it using the procedure `write`. In TURBO Pascal it is possible to mix reading from and writing to a file, provided the file is of some type other than `text`. However, at first it may be easier to only read from files opened with `reset` and only write to files opened with `rewrite`.

All files must be closed when a program is through reading from and/or writing to the file. For all types of files, the syntax for `close` is identical to the syntax that we described for text files.

*string names*

*reset  
rewrite*

*close*

---

## Standard Pascal—Opening Files (Optional)

In standard Pascal, file variable names for nontext files are listed in the program heading in the same manner as we described for text files in Chapter 13. Nontext files are opened for reading with the standard procedure `reset` and are opened for writing with the standard procedure `rewrite`. The usage is identical to the usage that we described for text files in Chapter 13, except that the file variable name must be declared as we described in this chapter in the section entitled “File Variables.”

A `write` statement may be used with any file that has been opened with the standard procedure `rewrite`. A `read` statement may be used with any file that has been opened with the standard procedure `reset`.

---

## Windows

Before we go on to discuss ways of reading from and writing to nontext files, we must first explicate one preliminary concept, namely, the notion of a *window*. (Windows are called *file pointers* in some books.) As we have already said, a file is a sequence of

*file pointer*



components all of the same type. Every file has a window that is positioned at exactly one of these components. If these components are integers, then the window is positioned at one integer. If the file is a file of records, then the window is positioned at one record. As the term “window” indicates, the program has access to (can “see”) only one component in the file, namely, the component at which the window is positioned. To read things in a file, the program must somehow move the window to the position of the component to be read. Similarly, when writing to a file, the program can only write at the current position of the window. That means that files can only be accessed one component (integer or record or whatever) at a time.

---

## read and write

The predefined procedure `write` may be used with files of any type, not just with files of type `text`. To set the scene for an example, suppose a program contains the following declarations:

```
type RealArray = array[1..100] of real;
var File1: file of integer;
    N, M: integer;
    File2: file of RealArray;
    B, C: RealArray;
```

Suppose further that the file variable names `File1` and `File2` have been associated with files and opened for writing. In `TURBO Pascal` this can be accomplished with calls to `assign` and `rewrite`. (In standard Pascal all that is needed are calls to `rewrite` and a correct program heading.)

In order to write the numbers 7 and 11 to the file `File1`, the following will suffice:

```
N := 7; M := 11;
write(File1, N, M)
```

The values of the array variables `B` and `C` can be written to `File2` with the statement

```
write(File2, B, C)
```

As with text files, the first argument to `write` is the file variable name of the file. This argument is followed by one or more arguments giving values to be written out. These other arguments must be variables of the component type. If the component type is `integer`, then all the variables whose values are written out must be of type `integer`. If the component type is an array type, then the variables whose values are written out must all be of that array type.

Some versions of Pascal, including `TURBO Pascal`, require that the values to be written out to a nontext file be given as *variables*. They cannot be given as literal con-

---

stants or as expressions other than variables. Hence, neither of the following should be used:

```
write(File1, 7, 11); {No good}
write(File1, 2 + 2)  {No good}
```

This requirement is one difference between the use of `write` with nontext files and with text files.

As with `write` the predefined procedure `read` can be used with files of any type. The first argument to `read` is a file variable name, and the remaining arguments are variables of the component type of the file. The call will set the values of the variables equal to as many of the components of the file as there are variables. After each value is read from the file, the file window is advanced to the next component.

The use of the procedures `write` and `read` with nontext files are essentially the same as they are for text files. However, the procedures `writeln` and `readln` do not work for files of types other than `text`. In fact, they do not make sense for files of types other than `text`, since such files are not divided into lines.

---

## TURBO Pascal Case Study

---

### Processing a File of Numeric Data

#### Problem Definition

As a simple example of the use of files, we will design a program that reads numbers from a file of reals, multiplies each number by 2, and then copies the result to a second file of reals. The file type is declared as follows:

```
type NumberFile = file of real;
```

The program will read from a file with string name 'OLD.DAT' and will write its output numbers to a new file with string name 'NEW.DAT'. Suppose that before the program is run, the file 'NEW.DAT' does not exist and that the file 'OLD.DAT' contains the components

```
1.1    2.2    3.3    4.4    5.5
```

After the program is run, the file 'OLD.DAT' will be unchanged, the file 'NEW.DAT' will have been created, and 'NEW.DAT' will contain the components

```
2.2    4.4    6.6    8.8    11.0
```

(Although the above displays might make you think you could read the numbers in the file using the editor, you cannot. The numbers are coded in machine-readable form and can only be read by a Pascal program.)

---

The program will use the file variable names `OldFile` and `NewFile` for the two files. After the following calls to `assign`, the files will always be referred to by their file variable names:

```
assign(OldFile, 'OLD.DAT'); assign(NewFile, 'NEW.DAT');
```

### Discussion

In order to copy a number from one file to another, a Pascal program must first read the number into a variable and then write the value of the variable to another file. Such a variable is usually called a *buffer variable*. If the buffer variable is named `Buffer`, the basic way to copy a number from `OldFile` to `NewFile` is

```
read(OldFile, Buffer);
write(NewFile, Buffer)
```

If we want to double the numbers, we simply double the value in `Buffer` before we have the program write it to the second file. So the basic outline of our algorithm is

### ALGORITHM

```
open OldFile with reset;
open NewFile with rewrite;
for each number in OldFile do the following:
begin
  Read(OldFile, Buffer);
  Buffer := 2 * Buffer;
  write(NewFile, Buffer)
end
```

### eof

Just as we did with text files, we can use the boolean `eof` to detect the end of the file being read from. So the loop in our algorithm can be implemented with a *while* loop that uses the boolean `eof(OldFile)`. The complete program is given in Figure 16.1.

## Pitfall

### Unexpected End of File

If a program is reading from the keyboard, it can ask the user whether there is more data or not. If it is reading from a file, there is no user to ask, and so the program must know when to stop reading data. If your program is not written so that it stops reading when the end of a file is reached, then the program will terminate abnormally when the file has been exhausted. Fortunately, the boolean `eof` can be used to detect the end of a nontext file. The boolean `eof` is used in the same way for nontext files as it is for text files. The use is illustrated in Figure 16.1.



```
program Double;
{Reads reals from the file OLD.DAT, multiplies each by 2, and writes
the result to the file NEW.DAT. If NEW.DAT does not exist, it is created;
if it already exists, the old contents are lost. The file OLD.DAT is not changed.}
type NumberFile = file of real;
var OldFile, NewFile: NumberFile;
    Buffer: real;

begin{Program}
  writeln('Doubling program started. ');
  assign(OldFile, 'OLD.DAT'); assign(NewFile, 'NEW.DAT');

  reset(OldFile);
  rewrite(NewFile);
  {The window is at the first component of OldFile; NewFile is blank.}

  while not eof(OldFile) do
    begin{while}
      read(OldFile, Buffer);
      Buffer := 2 * Buffer;
      write(NewFile, Buffer)
      {Up to but not including the position of the windows, the
      components of NewFile are those of OldFile multiplied by 2.}
    end; {while}

  close(OldFile); close(NewFile);
  writeln('End of program. ')
end. {Program}
```

**Figure 16.1**  
**Program using**  
**nontext files.**

---

## Self-Test Exercises

1. Give a suitable type declaration for a file that is to hold student records consisting of a name, a final exam score in the range 0 to 100, and a letter grade.
  2. Write a program to create a file of type *file of integer* and to write the numbers one through ten to the file.
  3. Write a program that displays to the screen the contents of a file of the type used in the program of the previous exercise.
  4. Write a program to search a file of integers to see whether it contains a particular integer. The particular integer should be read in from the keyboard.
-

**Program**

```
program BuildFile;
{Creates the file FileName and fills it with records.}
const FileName = 'RECORDS.DAT'; {String name of the file.}
      MaxLength = 30; {Maximum length for a name.}
type Spell = string[MaxLength];
      Employee = record
          Number: integer;
          Name: Spell;
          PayRate: real
      end;
      PayRecords = file of Employee;
var OneRecord: Employee;
    PayFile: PayRecords;

procedure ReadRecord(var OneRecord: Employee);
{Reads values into OneRecord. If Number field is
negative, it does not fill the remaining fields.}
begin {ReadRecord}
    with OneRecord do
        begin {with}
            writeln('Enter employee number: ');
            readln(Number);
            if Number >= 0 then
                begin {then}
                    writeln('Enter employee name: ');
                    readln(Name);
                    writeln('Enter rate of pay: ');
                    readln(PayRate);
                    writeln {to keep the screen neat}
                end {then}
            end {with}
        end; {ReadRecord}

begin {Program}
    assign(PayFile, FileName);
    rewrite(PayFile);

    writeln('Enter data for each record of ', FileName);
    writeln('To quit, enter a negative employee number. ');
```

**Figure 16.2**  
**Building a file of**  
**records**

```
ReadRecord (OneRecord);  
while OneRecord.Number >= 0 do  
  begin{while}  
    write (PayFile, OneRecord);  
    ReadRecord (OneRecord)  
  end; {while}  
  
close (PayFile);  
writeln (FileName, ' filled with records. ')  
end. {Program}
```

### Sample Dialogue

```
Enter data for each record of RECORDS.DAT  
To quit, enter a negative employee number.  
Enter employee number:  
198  
Enter employee name:  
Lee Iacocca  
Enter rate of pay:  
11057.69  
  
Enter employee number:  
101  
Enter employee name:  
Alan Greenspan  
Enter rate of pay:  
33.65  
  
Enter employee number:  
-1  
RECORDS.DAT filled with records.
```

**Figure 16.2**  
(continued)

---

## TURBO Pascal Example

---

### Creating a File of Records

The component type of a file may be a structured type, such as a record or array type. Files of records are common for fairly obvious reasons. Employee records, inventory stock records, student transcript records, and almost any other sort of “records” can be kept in a permanent form in secondary storage using a file of records.

Figure 16.2 contains a program that creates a file of records. Note that each file



component is a complete record and that it is written to the file as a single unit. The complete record `OneRecord` is written to the file `PayFile` with the statement

```
write(PayFile, OneRecord)
```

---

## Deciding What Type of File to Use

Since `integer` and `real` values can be stored in text files, there may seem to be no need for the file types *file of integer* and *file of real*. This is not quite true. When storing numbers in a file of type `text`, a type conversion is performed whenever a number is read from or written to the file. With the other file types, no type conversion is needed. Hence, if all the data in a file is of type `integer`, then a program will run more efficiently if the type of the files used is *file of integer* rather than `text`. In the case of the type `real`, accuracy is also an issue. When using a text file, each reading or writing of a value of type `real` can introduce inaccuracies in the value stored. This loss of accuracy is a result of the type conversion calculation. By using a file of type *file of real*, you avoid these type conversions and the inaccuracies they produce.

When choosing a file type, remember that the components of a file can be of a structured type. If your program needs to place an array of `real` values in secondary storage, then a *file of real* will work acceptably. However, a much simpler program can be written using a file with an array component type. For example, consider the following declarations:

```
type ArrayType = array[1..100] of real;
var A: ArrayType;
    DiskFile: file of ArrayType;
```

Once the file has been opened, the array can be placed in secondary storage with the one write statement displayed next.

```
write(DiskFile, A)
```

This is much simpler and easier to read than a *for* loop that separately copies each value from the array to a *file of real*.

## TURBO Pascal

### seek

In TURBO Pascal, programs can mix reading from and writing to the same nontext file. Hence, with nontext files, file modifications can sometimes be accomplished without using an additional temporary file, as we always needed to do when modifying text

---

files. The `reset` statement opens an existing file. The `rewrite` statement creates a new file. In either case the file can be both read from and written to by intermixing calls to `read` and `write`. To do this effectively, it is usually necessary to position the window with the `seek` command described in the next paragraph.

TURBO Pascal allows random access to files. In other words, the window can move backward as well as forward and can jump directly to any desired location. This movement is accomplished with the predefined procedure `seek`. For example, the following will move the window in the file named by the file variable `PayFile` to component number 5 in the file:

*seek*

```
seek(PayFile, 5)
```

The procedure `seek` takes two arguments; the first is a file variable name, and the second is an expression that evaluates to a nonnegative integer value. The call to `seek` moves the file window to the component whose number is specified by the integer expression. So the next call to `read` or `write` will apply to that component. *The components are numbered starting with 0 rather than 1.* Thus, the first component is component number 0, the next is component number 1, and so forth. The `seek` statement does not work for text files.

By way of example, suppose that `X` is a variable of some type called `Employee` and that `PayFile` is a file variable name for a file of component type `Employee`. To set the value of `X` equal to component number 5 in the file `PayFile`, you can use the following code:

```
seek(PayFile, 5);  
read(PayFile, X)
```

To change component number 9 to the value of `X`, the following code will suffice:

```
seek(PayFile, 9);  
write(PayFile, X)
```

Since each execution of `read` or `write` moves the file window, it may be necessary to include a call to `seek` before each call to either of these procedures.

You should not attempt to use `seek` to move the window more than one location beyond the last component. You are allowed to seek one component beyond the end of the file in order to add a component to the end of a file. The predefined function `filesize` can be used to find the end of a file. `filesize` returns the number of components in the file named by its file variable argument. For example,

*filesize*

```
seek(PayFile, filesize(PayFile))
```

will position the window in `PayFile` one location beyond the end of the file. The procedure `write` can then be used to add a component to the file.

Remember that file components are numbered starting with 0. So `filesize(PayFile)` returns a position number one more than that of the last file component. For example, if the file contains three components, they are numbered 0, 1, and 2; and `filesize(PayFile)` returns 3.

---

## TURBO Pascal Examples

---

### Random and Sequential File Access

The program in Figure 16.3 uses `seek` with a file of records. Because it uses `seek`, it need not search the file sequentially. The user specifies the number of the record that he or she wishes to see from the file `PayFile`. This number is read into the variable `Number`. After that, the window is positioned, and the record is read into the variable `OneRecord` with the two statements

```
seek(PayFile, Number);
read(PayFile, OneRecord);
```

This method of random access to files can be very powerful, but it does have one limitation: The number of the component within the file must be known. In the sample programs, each component is a record. The records are stored sequentially and num-

```
program ShowByFileNumber;
{Displays records of the file FileName.
User must specify the record numbers.}
const FileName = 'RECORDS.DAT'; {String name of file.}
      MaxLength = 30; {Maximum length for a name.}
type Spell = string[MaxLength];
      Employee = record
          Number: integer;
          Name: Spell;
          PayRate: real
      end;
      PayRecords = file of Employee;
var OneRecord: Employee;
    PayFile: PayRecords;
    Number, LastRecord: integer;

procedure ShowRecord(OneRecord: Employee);
begin {ShowRecord}
    with OneRecord do
        begin {with}
            writeln('Record contains:');
            writeln('Employee Number: ', Number);
            writeln('Name: ', Name);
            writeln('Rate of Pay: ', PayRate:8:2)
        end {with}
    end; {ShowRecord}
```

**Figure 16.3**  
Random access  
to a file.



```
begin{Program}
  writeln('This program will display any record in');
  writeln(FileName, ' given the record number.');
```

assign(PayFile, FileName);  
reset(PayFile);

LastRecord := filesize(PayFile) - 1;  
writeln('Enter a record number from 0 through ', LastRecord);  
writeln('Enter a negative number to quit.');

readln(Number);  
while Number >= 0 do  
 begin{while}  
 seek(PayFile, Number);  
 {The window is at component Number.}  
 read(PayFile, OneRecord);  
 ShowRecord(OneRecord);  
 writeln;  
 writeln('Enter record number. Enter a negative number to quit.');

readln(Number)  
 end; {while}

close(PayFile);  
writeln('End of program.')

end. {Program}

### Sample Dialogue

(Assuming the program in Figure 16.2 was run first.)

This program will display any record in  
RECORDS.DAT given the record number.

Enter a record number from 0 through 1  
Enter a negative number to quit.

1

Record contains:

Employee Number: 101

Name: Alan Greenspan

Rate of Pay: 33.65

Enter record number. Enter a negative number to quit.

0

Record contains:

Employee Number: 198

Name: Lee Iacocca

Rate of Pay: 11057.69

Enter record number. Enter a negative number to quit.

-1

End of program.

**Figure 16.3**  
(continued)

bered 0, 1, 2, and so forth. The location number must somehow be known before seek can be used to retrieve the record. In the sample file that we used in the last two programs, this number bears no relationship to either the employee number or the employee's name. If it is possible to reassign employee numbers so that the location numbers are used as the employee numbers, then seek can be used to recover a record given this number. However, this situation is often difficult to arrange. Moreover, it will not help in retrieving a file that is only specified by the employee's name. seek cannot directly locate a record given only the employee's name. Often the only way to locate the file of a named employee is to use a sequential search through the file, as illustrated in Figure 16.4.

### Program

```

program RecordSearch;
{Displays records of the file FileName. User specifies the employee's name.}
const FileName = 'RECORDS.DAT'; {String name of file.}
      MaxLength = 30; {Maximum length for a name.}
type Spell = string[MaxLength];
      Employee = record
          Number: integer;
          Name: Spell;
          PayRate: real;
      end;
      PayRecords = file of Employee;
var OneRecord: Employee;
    PayFile: PayRecords;
    Name: Spell;
    Found: boolean;
    Ans: char;

procedure ShowRecord(OneRecord: Employee);
begin{ShowRecord}
    with OneRecord do
        begin{with}
            writeln('Record contains:');
            writeln('Employee Number: ', Number);
            writeln('Name: ', Name);
            writeln('Rate of Pay: ', PayRate:8:2)
        end {with}
    end; {ShowRecord}

```

**Figure 16.4**  
Searching a file.

```
begin{Program}
  writeln('This program will display any record');
  writeln('given the employee's name. ');

  assign(PayFile, FileName);

  repeat
    writeln('Enter employee name: ');
    readln(Name);

    reset(PayFile);
    Found := false;
    while (not eof(PayFile)) and (not Found) do
      begin{while}
        read(PayFile, OneRecord);
        if OneRecord.Name = Name then
          Found := true
        end; {while}

    if Found then
      ShowRecord(OneRecord)
    else
      writeln('No record for ', Name);

    writeln('Again? (y/n) ');
    readln(Ans)
  until upcase(Ans) = 'N';

  close(PayFile);
  writeln('End of program. ')
end. {Program}
```

#### Sample Dialogue

```
This program will display any record
given the employee's name.
Enter employee name:
Alan Greenspan
Record contains:
Employee Number: 101
Name: Alan Greenspan
Rate of Pay: 33.65
Again? (y/n)
n
End of program.
```

**Figure 16.4**  
(continued)



## Files as Parameters to Procedures

Procedures may have parameters of any file type, but they must be *variable parameters*; they cannot be value parameters. The situation for other types of files is the same as that for text files. The following case study illustrates the use of file variable parameters.

---

### TURBO Pascal Case Study

---

#### Merging Two Files

##### Problem Definition

Data is often provided in the form of files that need to be processed at some central location. As a simple example, let us suppose that a company has two plants, which keep employee pay records of the form used in Figure 16.4. The files are sent from the plants to corporate headquarters, where they need to be merged into a single file containing all the records from the two files. In this section we will design a procedure to merge two such files into a third master file. We assume that both files are sorted by employee number from lowest to highest and that each file has a sentinel record that marks the end of the file. This sentinel record is assumed to have an employee number field that is larger than any actual employee number.

##### Discussion

The procedure that performs the merging will read the records in each file in the order that they occur, which is assumed to be lowest to highest employee number, and it will write them to the master file one at a time in the order lowest number to highest number. Things seem to be ordered in a fortuitous way. If the procedure reads the first record from each of the smaller files into two buffer variables, then we can be certain that one of the two buffer variables contains the lowest numbered record of all the records in the two files. Therefore, the procedure can write this record to the master file and then replace the just-written-out record with the next smallest record from the same file as the record just disposed of. At this point the two records in buffer variables include the next lowest number, and the process can be repeated. The process is illustrated in Figure 16.5. Since each file is terminated with a sentinel record that contains a very large number, we know that neither file will run out of records until the procedure empties both files and needs to compare the two sentinel values. At that point we can have the procedure end the process and then arbitrarily write either of the sentinel records at the end of the master file.

---

Let us use File1 and File2 as the file variables for the two files being merged and MasterFile as the file variable for the file to receive the merged list. The two buffer variables to hold one record from each file will be called Buffer1 and Buffer2. The files File1 and File2 are opened with reset, MasterFile is opened with rewrite, and then the following algorithm is executed to perform the merge:

*variables*

(continued, page 626)

(a)

File1	50	60	90	maxint
	J. Smith	A. Fratkin	R. Brandes	
	16.50	17.00	15.00	

Window

Buffer1


File2

70	80	maxint
T. Jones	D. Coleman	
17.00	19.00	

Window

Buffer2


MasterFile is empty.

(b)

File1	50	60	90	maxint
	J. Smith	A. Fratkin	R. Brandes	
	16.50	17.00	15.00	

Window

Buffer1

50
J. Smith
16.50

File2

70	80	maxint
T. Jones	D. Coleman	
17.00	19.00	

Window

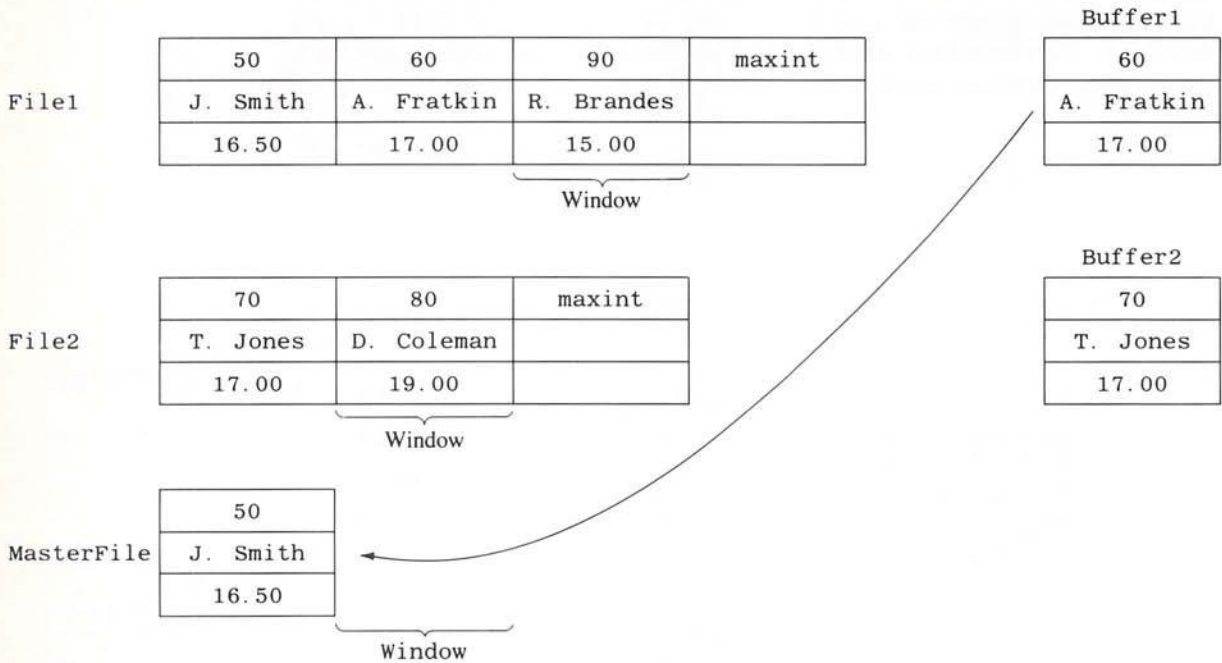
Buffer2

70
T. Jones
17.00

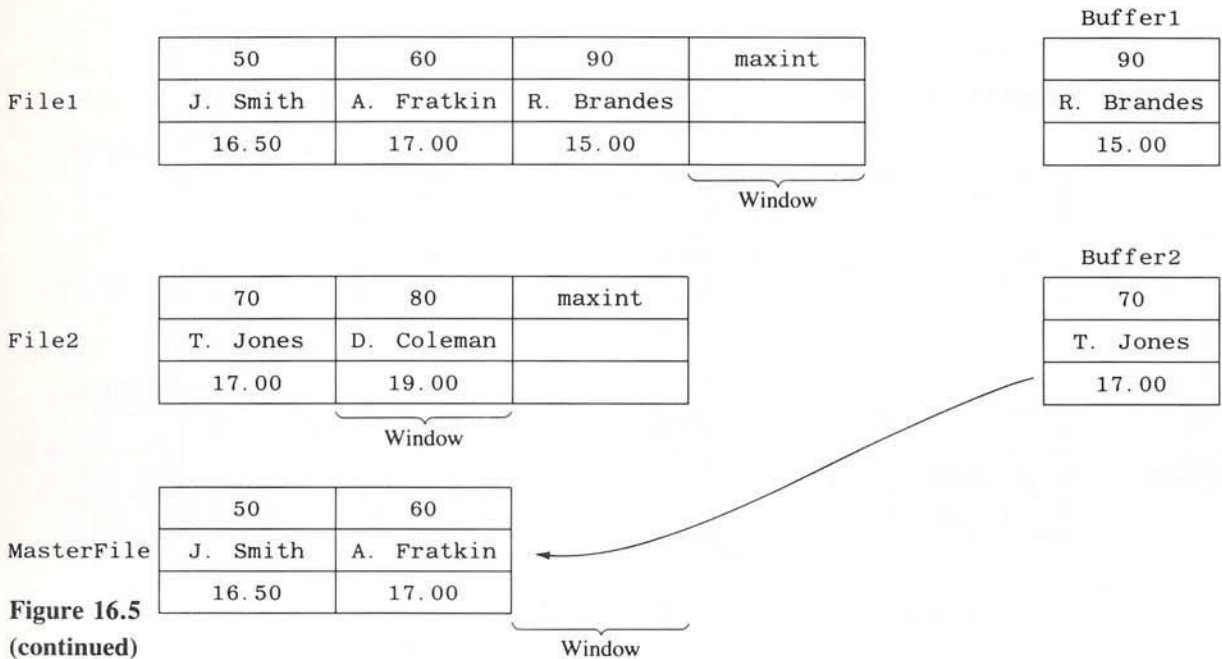
MasterFile ←

**Figure 16.5**  
Merging two files.

(c)



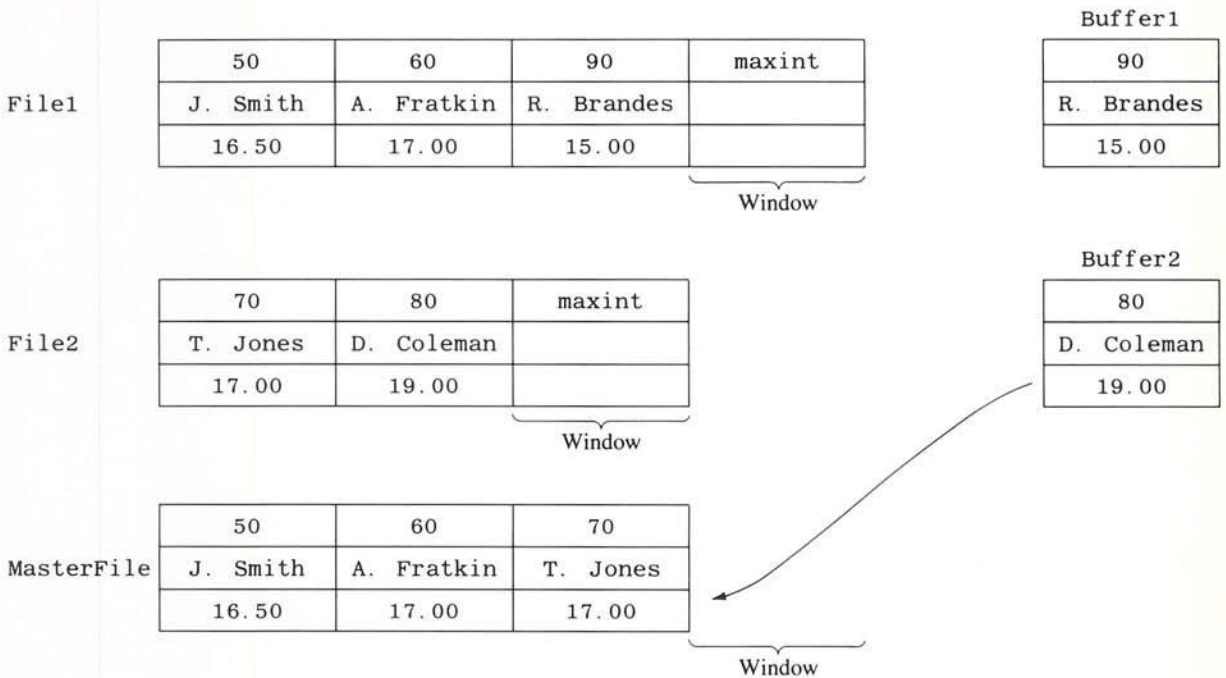
(d)



**Figure 16.5**  
(continued)



(e)



(f)

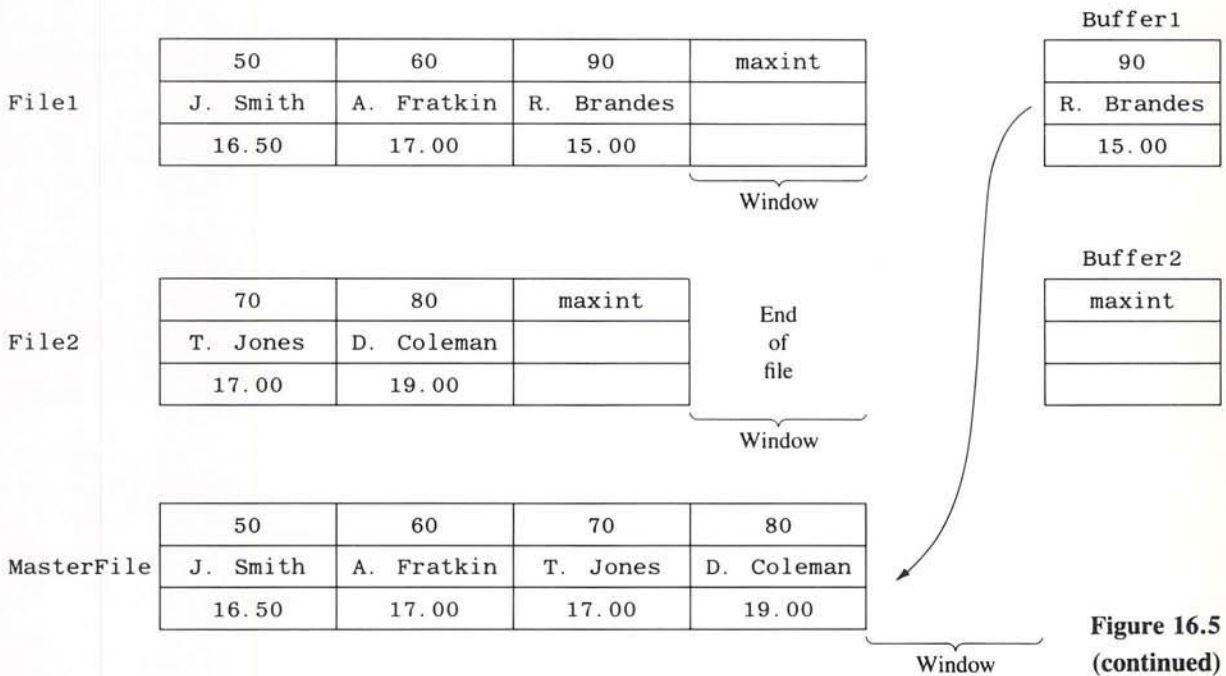
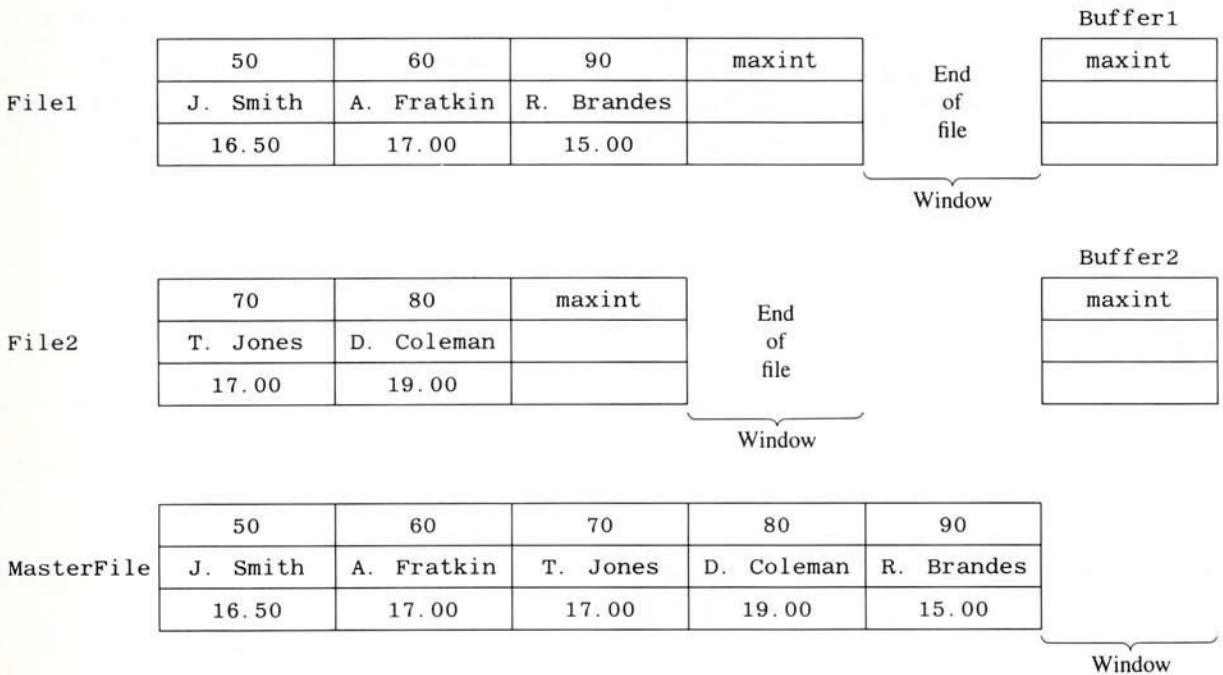


Figure 16.5  
(continued)

(g)



(h)

A sentinel record is added to MasterFile.

	50	60	70	80	90	maxint
MasterFile	J. Smith	A. Fratkin	T. Jones	D. Coleman	R. Brandes	
	16.50	17.00	17.00	19.00	15.00	

**Figure 16.5**  
(continued)

**ALGORITHM**

```

read(File1, Buffer1);
read(File2, Buffer2);
while (not eof(File1)) or (not eof(File2)) do
    begin
        Find out which of Buffer1 or Buffer2
        has the smaller number and write that record to MasterFile;
        Replace the record written with one from the same file.
    end
write(MasterFile, Buffer1); {Buffer1 is a sentinel record.}

```

The final procedure, embedded in a complete program, is shown in Figure 16.6. The body of the *while* loop is implemented as the procedure *CopySmaller*.

The procedure *Merge* can be quite inefficient for some files. If the records in one file are all copied before the other file is emptied, then the procedure will compare all the remaining records to the sentinel record. If the number of records remaining is large, this can be very time-consuming. A more efficient procedure would detect when it had reached the end of one of the files, and at that point would simply copy the records remaining in the other file into the master file. We leave the design and coding of such an efficient procedure as an exercise. (See Exercise 11.)

*efficiency*

```

program MergFiles;
{Merges the two files specified by the user into a third file with
records sorted by employee number. Precondition: FileName1 and FileName2
are each already sorted by employee number.}
const MaxLength = 30; {Maximum length for an employee name.}
type Spell = string[MaxLength];
   Employee = record
       Number: integer;
       Name: Spell;
       PayRate: real
   end;
   PayRecords = file of Employee;
var FileName1, FileName2, MasterName: string[20];
    File1, File2, MasterFile: PayRecords;

procedure CopySmaller(var Buffer1, Buffer2: Employee;
                     var File1, File2, MasterFile: PayRecords);
{Precondition: not eof(File1) or not eof(File2); Buffer1 and Buffer2 contain
the last records read from File1 and File2, respectively. Postcondition: The correct
record has been copied to MasterFile and replaced by the next record from the same file.}
begin{CopySmaller}
    if Buffer1.Number < Buffer2.Number then
        begin{Buffer1.Number < Buffer2.Number}
            write(MasterFile, Buffer1);
            read(File1, Buffer1)
        end {Buffer1.Number < Buffer2.Number}
    else
        begin{Buffer2.Number <= Buffer1.Number}
            write(MasterFile, Buffer2);
            read(File2, Buffer2)
        end {Buffer2.Number <= Buffer1.Number}
    end; {CopySmaller}

```

**Figure 16.6**  
Program to merge  
two files.



```
procedure Merge(var File1, File2, MasterFile: PayRecords);  
{Precondition: The files File1 and File2 are sorted by employee number, and  
each is terminated with a sentinel record whose Number field has the value maxint.  
Postcondition: MasterFile contains all the records from File1 and File2, they  
are sorted by employee number, and there is a sentinel record at the end.}  
var Buffer1, Buffer2: Employee;  
begin{Merge}  
    writeln('Merging started. ');  
  
    reset(File1);  
    reset(File2);  
    rewrite(MasterFile);  
  
    read(File1, Buffer1);  
    read(File2, Buffer2);  
    while (not eof(File1)) or (not eof(File2)) do  
        CopySmaller(Buffer1, Buffer2, File1, File2, MasterFile);  
    {Both eof(File1) and eof(File2) are true.}  
  
    {Buffer1 contains a sentinel record.}  
    write(MasterFile, Buffer1);  
  
    writeln('Merging completed. ')  
end; {Merge}  
  
begin{Program}  
    writeln('Enter the names of the two files');  
    writeln('to be merged (one per line): ');  
    readln(FileName1);  
    readln(FileName2);  
    assign(File1, FileName1);  
    assign(File2, FileName2);  
  
    writeln('Enter a name for the merged file: ');  
    readln(MasterName);  
    assign(MasterFile, MasterName);  
  
    Merge(File1, File2, MasterFile);  
  
    close(File1); close(File2); close(MasterFile);  
    writeln('End of Program')  
end. {Program}
```

**Figure 16.6**  
(continued)

---

---

## Summary of Problem Solving and Programming Techniques

- Data of almost any type, including record types, may be kept in secondary storage by using a file with that particular component type.
- Although numeric data can be stored in a text file, it is usually more efficient and clearer to store it in a file whose component type matches the type of the data.
- Nontext files are opened and named in the same way as text files.
- The procedures `writeln` and `readln` and the boolean `eoln` do not make sense for and so cannot be used with nontext files. The procedures `read` and `write`, as well as the boolean `eof`, can be used with other files in basically the same way that they are used with text files.
- The exact details of file handling will vary from one installation to another. Hence, file handling should be isolated into procedures in order to make any needed changes easy to carry out.

---

## Summary of Pascal Constructs

---

### Constructs Common to All Versions of Pascal

#### file types

Syntax:

*file of* <component type>

Example:

```
type Item =  
    record  
        Field1: integer;  
        Field2: array[1..10] of real  
    end;  
DataFile = file of Item;
```

Type of a file to hold components of type <component type>. The <component type> may be any type that does not involve a file type.

#### file variable declarations

Syntax:

**var** <file var>: <file type name or definition>;

Examples:

```
var File1: file of integer;  
    File2: DataFile;
```

---

Declaration of a file variable. (The sample type `DataFile` is defined in the previous example.)

### **reset**

Syntax:

```
reset (<file var>)
```

Example:

```
reset (File1)
```

Opens the file named by the file variable `<file var>` and positions the window at the first component in the file. The file named by `<file var>` must already exist.

### **rewrite**

Syntax:

```
rewrite (<file var>)
```

Example:

```
rewrite (File1)
```

Creates a new file named by the file variable `<file var>`, opens the file for writing and positions the window to receive the first component. If there already is a file whose file variable name is `<file var>`, then the contents of the old file are lost. (Since the file is initially empty, the only sensible first action is to write to the file, after which TURBO Pascal allows you to mix reading and writing to the file.)

### **read**

Syntax:

```
read (<file var>, <component type var>)
```

Example:

```
read (File1, X)
```

The predefined procedure `read` used with a file of type other than `text`. `<file var>` is the file variable name of the file, and `<component type var>` is a variable, or list of variables, of the component type of the file. There may be any number of variables. The value of `<component type var>` is set equal to the value of the component in the window and the window is advanced to the next component. If there is more than one variable to receive values, then a value is read into each variable in this way.

### **write**

Syntax:

```
write (<file var>, <component type var>)
```

Example:

```
write (File1, X)
```

---



The predefined procedure `write` used with a file of type other than `text`. `<file var>` is the file variable name of the file, and `<component type var>` is a variable, or list of variables, of the component type of the file. There may be any number of variables to be written out. The value of the component in the window is set equal to the value of the variable, and the window is advanced to the next component. If there is more than one variable to be written out, then this process is repeated for each such variable.

### **eof**

Syntax:

```
eof (<file var>)
```

Example:

```
eof (File1)
```

A boolean function that returns `true` if the window in the file named by the file variable `<file var>` is beyond the last component and `false` when a file component is in the window.

---

## **TURBO Pascal Constructs**

### **assign, close, erase, rename**

Same as described for text files in Chapter 13.

### **seek**

Syntax:

```
seek (<file var>, <index>)
```

Example:

```
seek (File1, 89)
```

Moves the window in the file named by the `<file var>` to component number `<index>`. The argument `<index>` is an expression that evaluates to a nonnegative integer. The components are numbered 0, 1, 2, 3, . . . . The `seek` procedure cannot be used with text files.

### **filesize**

Syntax:

```
filesize (<file variable>)
```

Example:

```
filesize (File1)
```

Returns the number of components in the file named by `<file variable>`. Returns zero if the file is empty.

---

**filepos**

Syntax:

```
filepos (<file variable>)
```

Example:

```
filepos (File1)
```

Returns the current position of the file window (file pointer) in the file named by <file variable>. The components are numbered 0, 1, 2, 3, . . . . filepos cannot be used with text files.

**truncate** (Optional)

Syntax:

```
truncate (<file variable>)
```

Example:

```
truncate (File1)
```

This procedure is available on DOS systems but may not be implemented in other versions of TURBO Pascal. This procedure will truncate the nontext file at the location of the window; that is, the component in the window and all following components will be discarded. The file must have already been opened with either `reset` or `rewrite` before `truncate` can be used. After a call to `truncate`, the file is left open, and the window is positioned at the end of the file. `truncate` cannot be used with text files.

---

## Exercises

### Self-Test Exercises

5. Write a boolean-valued function with two arguments, `F`, a file of type `PayRecords`, and `N`, an integer. The type `PayRecords` is defined in Figure 16.4. The function should return `true` if the file contains a record for employee number `N` and should return `false` if the file contains no such record.
6. Which of the following predefined TURBO Pascal functions and procedures may be used with text files only, nontext files only, or both types of files?

```
assign, reset, rewrite, read, readln, write,  
writeln, seek, eoln, eof, seekeoln, seek eof,  
rename, filesize, filepos, erase, and close.
```

### Interactive Exercises

7. Write a program that fills an array with 10 integers read from the keyboard and then stores the array value in a file of the type defined as follows:
-

```
type List = array[1..10] of integer;  
  AFile = file of List;
```

8. Write a program that fills an array from the file created by the program of the previous exercise and then displays the array values to the screen.

### Programming Exercises

9. Write a program to search an existing file of integers and find both the largest and the smallest integers in the file.
10. Write a program that changes the file RECORDS.DAT of Figure 16.2 so that each employee number is changed so that it equals the file component number of the record. For example, the first record should have the employee number changed to 0, the next to 1, and so forth. Your program should work no matter what records are placed in the file by the program in Figure 16.2.
11. Rewrite the procedure Merge in Figure 16.6 so that it is more efficient in the manner described in the chapter. The sentinel records are not essential in this approach, and so your program should be designed to work with files that do not contain a sentinel record at the end.
12. Write a program that reads 10 integer values from a *file of integer* into an array, sorts the array, and then writes the sorted list back into the same file, so that the effect is to sort the numbers in the file.
13. Write a program that sorts a file of type *file of integer* so that the numbers appear in numeric order from the smallest to the largest. The final sorted list should be in the same file as the one originally containing the integers. The program should work for any size file, and so the program cannot read the numbers into an array as in the previous exercise.
14. A record for describing a person is to consist of the following items: last name, initial of first name, sex, age, height, weight, and telephone number. The records are to be stored in an array indexed by integers in the range 1 to Limit. Limit is to be declared as a constant. Write a program that reads in up to Limit records from the keyboard, stores them in an array, writes them out to the screen, and then writes them to a file. The file should be a file of records, not a text file or a file of arrays. Your program should allow the possibility of fewer than Limit records being read in.
15. Write a program that reads the file created in the previous exercise, places the components into an array, sorts the records in the array according to the alphabetical order of the last names, writes the sorted records to the screen, and then copies the records back to the file in alphabetical order.
16. Modify your program from Exercise 14 so that the second and succeeding times that it is run, the contents of the file are copied into an array; the user then has the option of either clearing all records and starting over or adding more records to those already in the array. Modify your program further to allow the user to delete individual records by specifying the last name for the record to be deleted. The user should also have the option of clearing all records without having to specify every name. Modify



your program further to allow the user to see all records of a given category (such as all records for individuals between two specified ages) on the screen. All manipulations should be done with the array. When the user is finished, the program should copy the modified collection of records back to the file so that the file then contains the same records as the array.

17. Redo the previous assignment, but this time have your program deal directly with the file and not use an array.

18. Write a program for a computerized dating service. The information on individuals should be kept in a file of records and should include all the information described in Exercise 14, plus other information, such as a hobbies, favorite color, and so forth. A user should be able to request a list of all dates that satisfy the user's specifications. Include a "best match" option that finds the date that is best suited to the user based only on the user's own record. Use a file of records.

19. Write a program to sort a file of records of the type described in Exercise 14. The files are to be arranged into alphabetical order according to last names.

20. Write a program to keep track of airline flight reservations and seat reservations. Allow any number of flights. Display seating plans as described in Exercise 19 of Chapter 10. The program should be able to add or delete flights. It should keep track of who is in what seat, as well as which seats are reserved. Keep the information in a file so that the program can be rerun and the information will be as it was left the last time the program was run. Use some file type other than `text`.

21. Write a program that fills a two-dimensional array of characters with a pattern typed in from the keyboard and then echoes back the pattern on the screen. Next write a program that uses a file whose component type is the type of the two-dimensional array. This program should store one array for each letter of the alphabet. When displayed, the array for each letter should display that letter as a large block letter. Finally, write a program that uses this file of arrays to read a word from the keyboard and to echo it back to the screen in block letters, one letter at a time.

22. If you are enrolled in a programming (or other) course, write a program that will serve as a record keeper for grades in your course. The program should allow the entry, display, and changing of any particular grade, such as a particular quiz or exam. If there is a formula for the final numeric grade, the program should calculate it. The program should allow the user to ask the class average for any particular grade, such as a quiz or exam grade. Use a file of records.

---



## *Dynamic Data Structures*

“You are sad,” the Knight said in anxious tone: “let me sing you a song to comfort you.”

“Is it very long?” Alice asked, for she had heard a good deal of poetry that day.

“It’s long,” said the Knight, “but it’s very, *very* beautiful. Everybody that hears me sing it—either it brings the *tears* into their eyes, or else—“

“Or else what?” said Alice, for the Knight had made a sudden pause.

“Or else it doesn’t, you know. The name of the song is called ‘*Haddocks’ Eyes*.’”

“Oh, that’s the name of the song, is it?” Alice said, trying to feel interested.

“No, you don’t understand,” the Knight said, looking a little vexed. “That’s what the name is *called*. The name really is ‘*The Aged Aged Man*.’”

“Then I ought to have said ‘That’s what the *song* is called’?” Alice corrected herself.

“No, you oughtn’t: that’s quite another thing! The *song* is called ‘*Ways and Means*’: but that’s only what it’s *called*, you know!”

“Well, what *is* the song, then?” said Alice, who was by this time completely bewildered.

“I was coming to that,” the Knight said. “The song really *is* ‘*A-sitting On A Gate*’: and the tune’s my own invention.”

*Lewis Carroll, Through the Looking-Glass*

## Chapter Contents

The Notion of a Pointer	Pitfall—Testing for the Empty List
Pascal Pointers and Dynamic Variables	Case Study—Using a Linked List to Sort a File
Manipulating Pointers	Stacks
Nodes	dispose (Optional)
The Pointer nil	Implementation (Optional)
Pitfall—Forgetting That nil Is a Pointer	TURBO Pascal—mark and release (Optional)
Linked Lists—An Example of Pointer Use	Doubly Linked Lists—A Variation on Simple Linked Lists
Case Study—Building a Linked List	Trees
The Empty List	Case Study—Building a Search Tree
Pitfall—Losing Nodes	Summary of Problem Solving and Programming Techniques
Case Study—Tools for Manipulating a Linked List	Summary of Pascal Constructs
Self-Test Exercises	Exercises
Pointer Functions	References for Further Reading

A *static data structure* is one whose structure is completely specified at the time the program is written and which cannot be changed by the program. Most of the data structures we have seen thus far, such as arrays and simple records, are static data structures. The values of the various components in these structures may change, but the structures themselves do not change. A program cannot change the number of items in an array or the number of indexes for the array. The structure is fixed.

*Dynamic data structures* may have their structure changed by the program. They may expand and contract in size as the program is executed, and as we will see in this chapter, the program can even change the manner in which the data is organized. In this chapter we will introduce a construct called a *pointer* and show how pointers can be used to construct a wide variety of dynamic data structures including *lists*, *stacks*, and a new data structure called a *tree*.



---

## The Notion of a Pointer

A *pointer* is, quite plainly and simply, something that points. That definition is very abstract. To make it concrete, we will give it a geometric interpretation in terms of figures drawn on paper. These figures will always be configurations of boxes connected by arrows. In these figures a pointer is represented by an arrow. Each of these arrows, or pointers, usually points to an object called a *node*. In these drawings a node is represented by a box, of any shape, in which things may be written or, more abstractly, in which data may be stored. For example, a list of records might be represented as nodes and pointers in the manner of Figure 17.1. In this case each record contains a number and a letter, and so the records might be student numbers and final course grades.

Before we present any more discussion of dynamic data structures, we will stop and discuss the Pascal syntax for pointers and nodes. This will allow us to express our algorithms as Pascal programs.

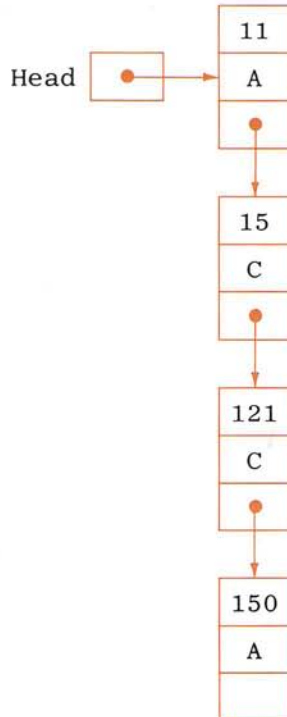
---

## Pascal Pointers and Dynamic Variables

In Pascal there is a special class of variables called *dynamic variables*. These dynamic variables are designed to be used as the nodes in dynamic data structures such as the

*nodes*

*dynamic  
variables*



**Figure 17.1**  
A data structure  
containing  
pointers.

list in Figure 17.1. In many ways dynamic variables are like the ordinary variables we have been using up until now. Like other variables, dynamic variables have a type. Dynamic variables may be of almost any type. The only exception is that dynamic variables of types involving files are not allowed. Dynamic variables can be assigned a value by means of an assignment operator, or a `read` statement, or by any other means that the value of an ordinary variable can be set. Similarly, the value in a dynamic variable can be accessed in any of the ways that the value of an ordinary variable can be accessed: by a `write` statement, by being an actual parameter to a procedure, or by any other means that the value of an ordinary variable can be accessed.

Dynamic variables differ from ordinary variables in only two ways. First, they may be created and destroyed by the program, and hence the number of such dynamic variables need not—indeed cannot—be determined at the time the program is written. Second, they have no names in Pascal; there are no identifiers that name them in the way that ordinary variables are named by identifiers. For these reasons, dynamic variables are not declared.

*pointer  
variables*

In order to refer to a dynamic variable, Pascal uses another type of variable called a *pointer variable*. Pointer variables are declared and do have identifiers associated with them. Pointer variables also have a type associated with them, and this type specifies the type of the dynamic variables with which they can be used. The value of a pointer variable is a pointer, and a pointer may point to a dynamic variable of the appropriate type. In this way, a dynamic variable may be referred to indirectly by giving a pointer that points to the dynamic variable. Typically, this is done by giving a pointer variable whose value points to the dynamic variable.

As an example, suppose we have the following record type declared in a program:

```
type Student = record
    Number: integer;
    Grade: char
end;
```

The program can have dynamic variables of type `Student`. They are not declared, but pointer variables, which can hold pointers that point to them, are declared. The following declares the variable `P` to be a pointer variable whose values are pointers to dynamic variables of type `Student`:

```
var P: ↑ Student;
```

The symbol `↑` is the “up-arrow” symbol. (It may look a bit different on some screens. A common variant is `^`, called a *circumflex* and looking like an arrow without the shaft. Some systems even use the totally different symbol `@` in place of `↑`.)

*domain  
type*

The type of the dynamic variables, such as `Student` in this example, is sometimes called the *domain type* of the pointer. The variable `P` can only contain pointers to dynamic variables of type `Student`. To hold a pointer to a dynamic variable of some other type, for instance `real`, requires a different pointer variable of the type `↑ real`. Dynamic variables and pointer variables both have types and, in order to hold a pointer to a dynamic variable, the type of the pointer variable must match that of the dynamic variable.

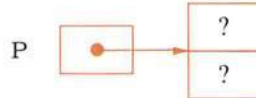
```

type Student = record
    Number: integer;
    Grade: char
end;
var P: ↑ Student;

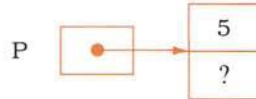
```



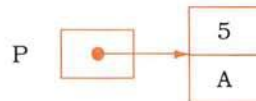
```
new (P)
```



```
P ↑ . Number := 5
```



```
P ↑ . Grade := 'A'
```



**Figure 17.2**  
Use of pointer  
variables and  
dynamic variables.

Of course, a pointer variable is of no use unless there is something for it to point to. In order to create a dynamic variable, the standard procedure *new* is used. For example, suppose that *P* is as previously declared, and consider the following statement:

```
new (P)
```

This will create a new dynamic variable of type *Student* and set the value of *P* equal to a pointer that points to this new dynamic variable. This dynamic variable can then be referred to as “the thing pointed to by *P*.” In Pascal the phrase “the thing pointed to by *P*” is denoted *P* ↑. In order to set the value of the component field *Number* in this new dynamic variable, we can use an assignment statement, like so:

```
P ↑ . Number := 5
```

This is illustrated in Figure 17.2. As shown there, the *Grade* field can be set in a similar way.

The values of the component fields of this dynamic variable can be written to the screen in the usual way:

```
writeln (P ↑ . Number, P ↑ . Grade)
```



Similarly,  $P \uparrow$  can be used anywhere else that it is appropriate to use a record variable of type `Student`.

## Manipulating Pointers

Before going any further, we had best stop and clarify the common English syntax used by programmers when discussing pointers and pointer variables. In the preceding example,  $P$  was a pointer variable. Technically speaking, it does not point to anything. It has values that are pointers, and these pointers point to dynamic variables. It may help to think of the pointer as an arrow and the pointer variable as something or somebody that can hold one pointer arrow at a time. This distinction between a pointer and a pointer variable is sometimes important. However, we will follow common usage and will usually blur this distinction. We will usually write, for example, “ $P$  points to a dynamic variable” when we really mean “the value of  $P$  points to a dynamic variable.”

### Program

```

program Test(input, output);

type Student = record
    Number: integer;
    Grade: char
end;
    StuPointer =  $\uparrow$ Student;
var P1, P2: StuPointer;

begin{Program}
    new (P1);
    P1 $\uparrow$ .Number := 1;
    P1 $\uparrow$ .Grade := 'A';
    writeln(P1 $\uparrow$ .Number, P1 $\uparrow$ .Grade);
    new (P2);
    P2 $\uparrow$ .Number := 2;
    P2 $\uparrow$ .Grade := 'B';
    writeln(P2 $\uparrow$ .Number, P2 $\uparrow$ .Grade);
    P1 := P2;
    P2 $\uparrow$ .Number := 3;
    P2 $\uparrow$ .Grade := 'C';
    writeln(P1 $\uparrow$ .Number, P1 $\uparrow$ .Grade, P2 $\uparrow$ .Number, P2 $\uparrow$ .Grade)
end. {Program}

```

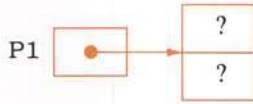
### Output

**Figure 17.3**     1A  
**Program that**     2B  
**illustrates pointers.**     3C 3C

(a)

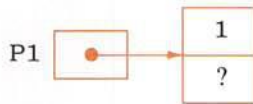


new (P1)



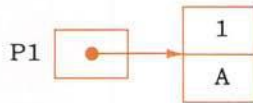
(b)

P1 ↑ . Number := 1



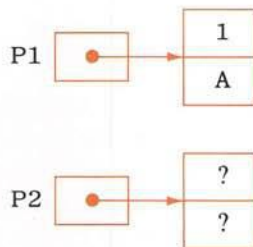
(c)

P1 ↑ . Grade := 'A'



(d)

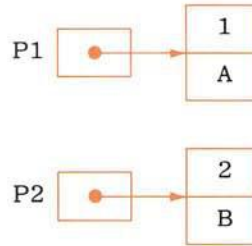
new (P2)



(e)

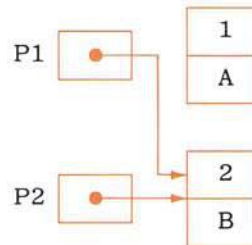
P2 ↑ . Number := 2;

P2 ↑ . Grade := 'B'



(f)

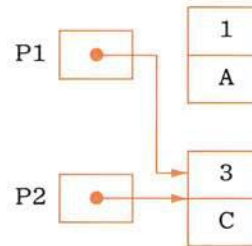
P1 := P2



(g)

P2 ↑ . Number := 3;

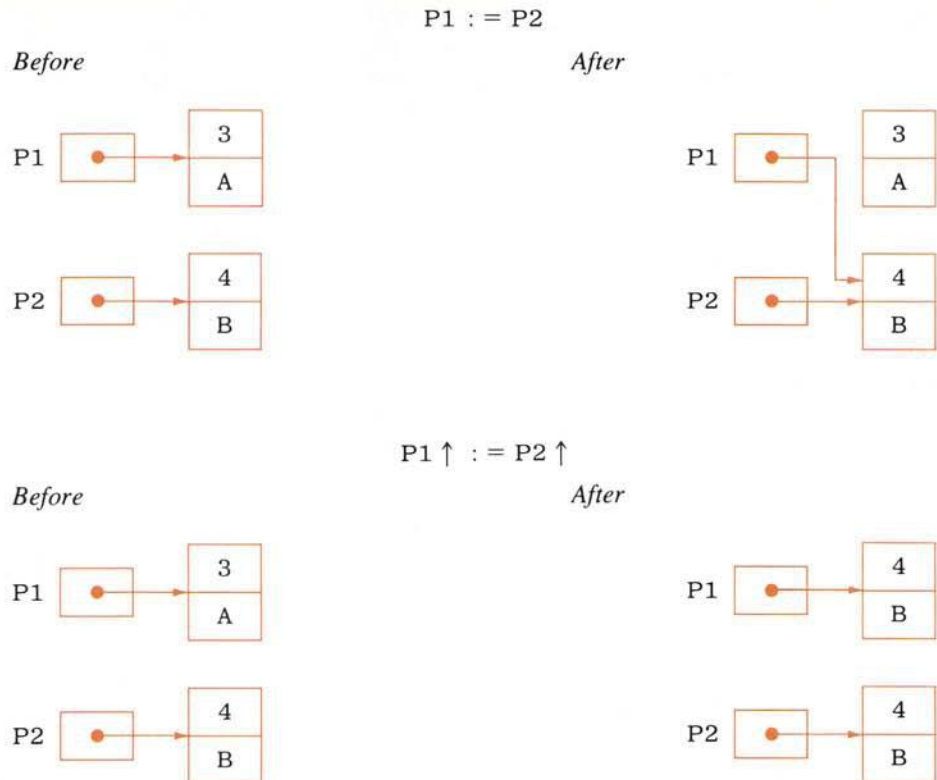
P2 ↑ . Grade := 'C'



**Figure 17.4**  
Explanation of  
Figure 17.3.

*:= and  
pointers*

Dynamic variables may have more than one pointer that is pointing to them. Also, pointers may be changed so that they point to different dynamic variables at different times. These changes are accomplished with the assignment operator `:=`. Technically speaking, there is nothing new involved. The assignment operator works with pointer variables in exactly the same way that it does with the other types of variables we have seen. However, interpreting the result can be a bit subtle. A sample program will help to illustrate the concepts. The workings of the program in Figure 17.3 are illustrated in Figure 17.4.



**Figure 17.5**  
The distinction  
between  
 $P1$  and  $P1 \uparrow$ .

In Figure 17.3 the statement

$P1 := P2$

changes the value of  $P1$  to that of  $P2$ . The values of  $P1$  and  $P2$  are pointers, and the only property a pointer has is that it points to something. Hence, the effect of this statement is to give  $P1$  a value that points to the same dynamic variable as the value of  $P2$  does. Stated this way, it makes it sound as though the two pointers in Figure 17.4(f) are the same pointer, and in a sense they are, just as two values of the number 5 stored in two different integer variables are the same integer 5. Those who find this too confusing, or too philosophical for their liking, can instead remember the more prosaic rule that if  $X$  and  $Y$  are pointer variables, then the statement

$X := Y$

changes  $X$  so that it points to the same thing that  $Y$  is currently pointing to.

When dealing with pointer variables, the distinction between a pointer variable,  $P1$ , for example, and the thing it points to,  $P1 \uparrow$ , is very important. When using the assignment operator, always be sure you check to see that you are referring to objects of the appropriate type. The distinction is illustrated in Figure 17.5.



---

## Nodes

The program in the previous section is a toy program. Nobody would use it for anything other than a learning aid. In fact, dynamic variables of type `Student` have almost no uses. Like the type `Student`, the type of a dynamic variable is invariably some sort of record type. However, the record type of a dynamic variable normally contains at least one field that is of a pointer type. By way of example, consider the data structure in Figure 17.1. The nodes in that data structure would be represented by records with three fields: two are of the types `integer` and `char`, the same as the type `Student`, but there is also one more field of a pointer type that can point to such nodes. The Pascal type declarations, as well as the declaration of the pointer variable `Head`, are shown at the end of this paragraph. The identifier `Node` can be replaced by any other identifier, but since the record represents a node, there is a tendency to call the type `Node`.

*type  
declaration*

```
type NPointer = ↑Node;
   Node = record
       Number: integer;
       Grade: char;
       Link: NPointer
   end;
var Head: NPointer;
```

This declaration is blatantly circular. `NPointer` is defined in terms of `Node`, and `Node` is defined in terms of `NPointer`. As it turns out, there is nothing wrong with this circularity, and it is allowed in Pascal. One indication that this definition is not logically inconsistent is the fact that we can draw pictures representing such structures. Figure 17.1 is one such picture. This is fortunate, since we must use some sort of circularity if we are to have data structures of this kind. After all, we want each node to contain a pointer to other nodes of the same type. If this is to be the situation, then the straightforward definition of the node type must refer to the pointers, and the straightforward definition of the pointer type must refer to the nodes. (In an attempt to avoid circularity, one clever programmer suggested defining a “pointer” as an arrow that points to “anything.” But alas, the programmer’s definition of “anything” referred to nodes, and the definition of “nodes” referred to pointers.) This is yet another example of how circular definitions can be not only meaningful, but also extremely useful in computer programming.

Pascal does require that the pointer type definition precede the associated node type definition; hence, the definitions of the types `NPointer` and `Node` must be in the order shown.

We now have pointers inside of records, and have these pointers pointing to records that contain pointers, and so forth. In these situations the syntax can sometimes get involved, but in all cases the syntax follows those few rules that we have described for pointers and records. As an illustration, suppose the declarations are as above, the situation is as diagrammed in Figure 17.1, and we want to change the value of the

---

Number field of the second node to 23; in other words, we want to change the 15 to a 23. One way to accomplish this is with the following statement:

```
Head↑.Link↑.Number := 23
```

To understand the expression on the left-hand side of the assignment operator, read it carefully from left to right. *Head* is a pointer; *Head*↑ is the thing it points to, namely the node (dynamic variable) containing 11. (This node can be referred to as *Head*↑, “but that’s only what it’s called, you know!” What it really is is the first node.) This node, referred to by *Head*↑, is a record, and the field of this record that contains a pointer is called *Link*, and so *Head*↑. *Link* is the name of a pointer that points to the node containing 15. Since *Head*↑. *Link* is the name of a pointer that points to the node containing 15, *Head*↑. *Link*↑ is a name for the node itself. Finally, *Head*↑. *Link*↑. *Number* is a name for the *Number* field of the node containing 15, and the assignment statement changes its value to 23. One can usually avoid such long expressions involving pointers, but occasionally they are useful, and they are a good test of whether or not you understand the syntax and semantics of pointers.

---

## The Pointer *nil*

There is one last Pascal construct that we need to describe before going on to discuss the applications of pointers. That is the constant *nil*. The constant *nil* is a predefined constant in the same sense that *maxint* is a predefined constant, although *nil* is of a different type than *maxint*. The constant *nil* is used to give a value to pointer variables that do not point to anything. As such, it can be used as a kind of end marker. For example, in the data structure shown in Figure 17.1, it would be reasonable to set the pointer field of the last node equal to *nil*. Then the program could test for the end of the list by checking to see if the pointer field equals *nil*. The usage of *nil* as an end marker is illustrated in the next section.

---

## Pitfall

### Forgetting That *nil* Is a Pointer

To avoid syntax errors, remember that *nil* is a pointer and not the thing pointed to. For example, suppose the declarations are as we described previously, and suppose the situation is as diagrammed in Figure 17.6(a); in that figure the pointer *Last* is of the same type as *Head* and has been positioned by some other part of the program. In order to set the pointer field of the last node to *nil*, the Pascal statement would be

```
Last↑.Link := nil
```

The effect of this assignment is diagrammed in Figure 17.6. Before this assignment, one pointer field had no value. Afterward it has the value *nil*. Notice that the type of the expression on the left-hand side of the assignment operator is a pointer type, namely the type *NPointer*. It is not a node type and so does not end with  $\uparrow$ .

The type of *nil* is a bit unorthodox, since its type can be that of a pointer to a dynamic variable of any type. To make this seem less strange, it should be pointed out (after a while the pun is unavoidable or perhaps just irresistible; anyway, it should be pointed out) that *nil* does not point to anything and, since it is a constant, it cannot be changed so that it does point to something.

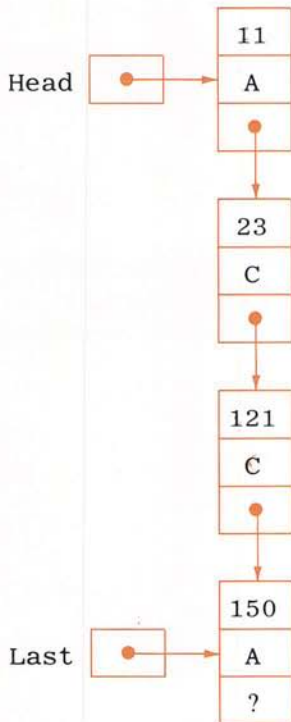
*type  
of nil*

## Linked Lists—An Example of Pointer Use

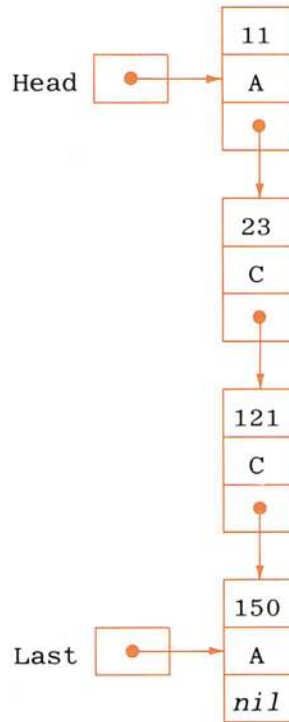
Structures like those in Figures 17.1 and 17.6 are called *linked lists*. The pointer *Head* (in either of these figures) is not part of the linked list, but is inevitably present when a linked list is being manipulated. A linked list consists of nodes, each of which has one

*head*

(a) Before



(b) After



**Figure 17.6**  
**Last  $\uparrow$  .Link := nil.**



pointer field. The pointers are set so that they order the nodes into a list. There is always one node called the *head* such that if you follow the arrows starting from that node, you will pass through each node exactly once. To put it less formally, the head is the first node in the list. The pointers called Head in these figures point to the heads of the linked lists given in the figures. (Do not confuse the pointer called Head with the head of the list. The first node in the list is called the head. The pointer variable is named Head, *not* because it *is* the head, but because it *points to* the head.)

Linked lists are used for many of the same things that arrays are used for, namely for storing lists of data. As we will see, a program can change the size of a linked list. Also, it is easy to insert or delete nodes in a linked list. For these reasons, linked lists are preferable to arrays for some applications.

## Case Study

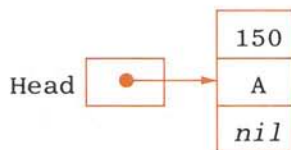
### Building a Linked List

As a warmup exercise, let us consider how we might construct the start of a linked list—that is, how we might make a short linked list consisting of one node. We will use nodes of the same type as we have been discussing. For reference, we repeat the type declarations.

```
type NPointer = ↑Node;
   Node = record
       Number: integer;
       Grade: char;
       Link: NPointer
   end;
var Head: NPointer;
```

To create a node we use the procedure `new`, set the two data fields, and then, since this is the last node as well as the first node, we set the `Link` field equal to `nil` in order to mark the end of the list. The following code will accomplish our goal and produce the short, one-node list displayed:

```
new(Head);
Head↑.Number := 150;
Head↑.Grade := 'A';
Head↑.Link := nil
```



adding  
nodes

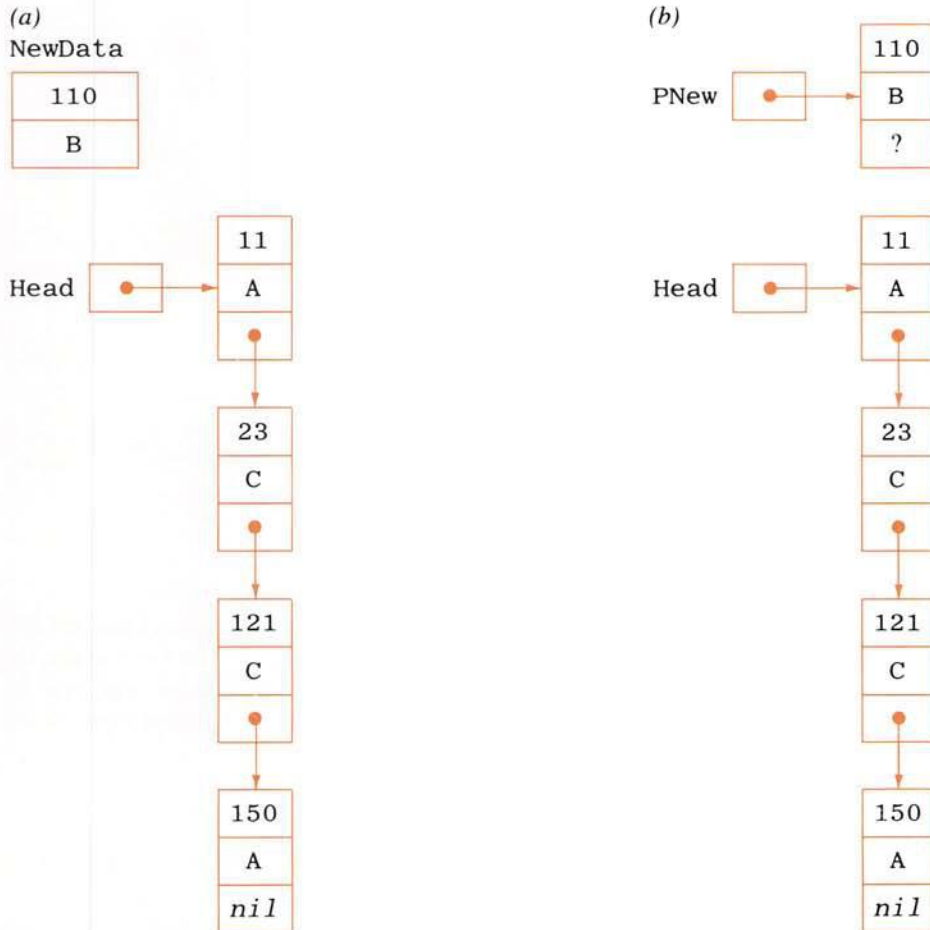
Our one-node list was built in a purely ad hoc way. In order to have a large linked list, a program must be able to add nodes to the linked list in a systematic way. We next

describe one simple way to insert nodes in a linked list. It will turn out that the procedure will work even if we start with an empty list. However, the process is clearer if we first assume that the list already has at least one node in it.

### Problem Definition

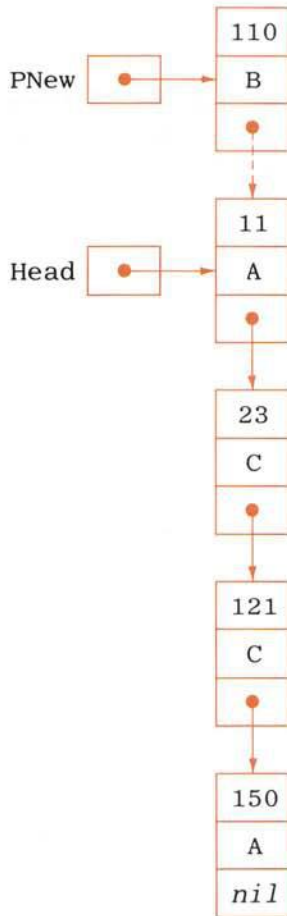
We want a procedure to insert new data in a linked list of the type shown in Figure 17.7(a). The linked list will be given by a pointer *Head* pointing to the head of the list. The data will be given by a record variable called *NewData* of the type *Data*, defined as follows:

```
type Data = record
    Number: integer;
    Grade: char
end;
```

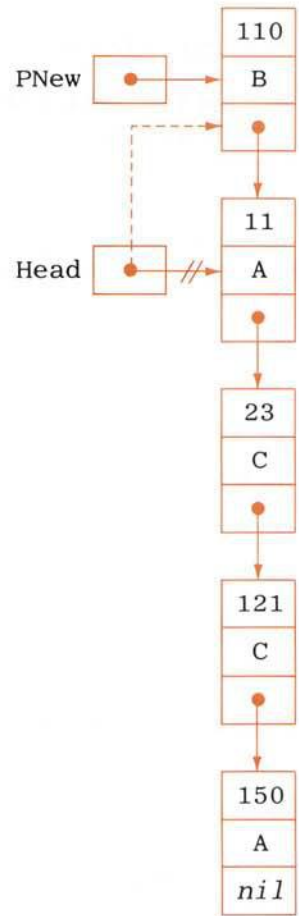


**Figure 17.7**  
Inserting a node at  
the head of a list.

(c)  
 $\text{PNew} \uparrow . \text{Link} := \text{Head}$



(d)  
 $\text{Head} := \text{PNew}$



**Figure 17.7**  
 (continued)

The procedure will change the list so that a new node with the given data is inserted and will redirect Head so that it points to the head of the modified list. For this problem we will not require that the list be in any particular order, and so the new node may be inserted anyplace in the list that is convenient. The task to be accomplished can be summarized as follows:

*Precondition: Head is pointing to the head of a linked list and NewData contains data for a new node.*

*Postcondition: A node, containing the data in NewData, has been added to the linked list; Head is pointing to the head of this enlarged list.*



## Discussion

In order to insert the data into the linked list, the procedure will need to use new to create a new node. The data is then copied into the new node, and the new node is inserted at the head of the list. Since dynamic variables have no names, we must use a local pointer variable to point to this node. If we call the local pointer variable PNew, then the new node can be referred to as PNew  $\uparrow$ . The complete process can be summarized as follows:

1. Create a new dynamic variable pointed to by PNew;
2. Place the data in this dynamic variable (which can be referred to as PNew  $\uparrow$ );
3. Make PNew  $\uparrow$  point to the head (first node) of the original linked list;
4. Make Head point to PNew  $\uparrow$  (i.e., the new dynamic variable).

### ALGORITHM

Figure 17.7 gives the algorithm in diagrammatic form. Steps 3 and 4 can be expressed by the two Pascal assignment statements given below. The complete procedure is given in Figure 17.8.

```
PNew  $\uparrow$ .Link := Head;
Head := PNew
```

```
type NPointer =  $\uparrow$ Node;
   Node = record
       Number: integer;
       Grade: char;
       Link: NPointer
   end;
   Data = record
       Number: integer;
       Grade: char
   end;

procedure HeadInsert(NewData: Data; var Head: NPointer);
{Inserts a node containing the data in the record NewData
at the head of the linked list headed by Head  $\uparrow$ .}
var PNew: NPointer;
begin{HeadInsert}
    new(PNew);    {Creates a new node to hold NewData.}
    PNew  $\uparrow$ .Number := NewData.Number;
    PNew  $\uparrow$ .Grade := NewData.Grade;
    PNew  $\uparrow$ .Link := Head;
    {Places the new node at the head of the list.}
    Head := PNew
    {Moves Head so it points to the new head of the list.}
end; {HeadInsert}
```

**Figure 17.8**  
Procedure to add  
a node to a linked  
list.

## The Empty List

A linked list is named by naming a pointer that points to the head of the list. To specify an empty list, the normal thing to do is to set this pointer equal to *nil*:

```
Head := nil
```

Whenever you design a procedure for manipulating a linked list, you should always check to see if it works on the empty list. If it does not, then it may be possible to add a special case for the empty list. If you cannot design the procedure to apply to the empty list, then the program must be designed to handle empty lists in some other way or to avoid them completely. One way to avoid empty lists is to add a *dummy node* that contains no real data but marks the end of the list and is never deleted.

Fortunately, the empty list can often be treated just like any other list. For example, the procedure `HeadInsert` in Figure 17.8 was designed with nonempty lists as the model, but a check will show that it works for the empty list as well.

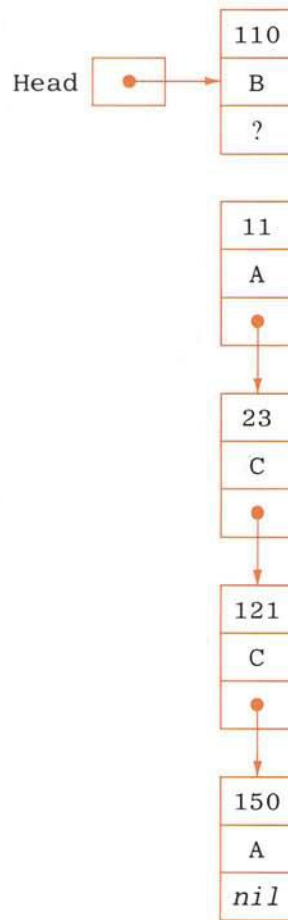
### Pitfall

#### Losing Nodes

You might be tempted to write the procedure `HeadInsert` using the pointer variable `Head` directly, instead of using the local pointer variable `PNew` to construct a new node. If we were to try, we might start the procedure as follows:

```
new(Head); {Creates a new node to hold NewData.}
Head↑.Number := NewData.Number;
Head↑.Grade := NewData.Grade;
```

At this point, the new node is constructed, contains the correct data, and is pointed to by the pointer `Head`, all as it is supposed to be. All that is left to do is to attach the rest of the list to this node by setting the pointer field `Head↑.Link` so that it points to what was formerly the first node of the list. Figure 17.9 shows the situation in the case where the new data values are 110 and 'B'. The diagram reveals the problem. If we proceed in this way, then unfortunately there would be nothing pointing to the node containing 11. Since there is no named pointer that is pointing to it or to any of the nodes below it, all those nodes are lost. There is no way that the program can reference them. It cannot make a pointer point to any of the nodes, nor can it access the data in those nodes, nor can it do anything else to the nodes. It simply has no way of referring to these nodes. To avoid such lost nodes, the program must always keep some pointer pointing to the head of the list, usually the pointer in a pointer variable like `Head`.



**Figure 17.9**  
Lost nodes.

## Case Study

### Tools for Manipulating a Linked List

In this section we will design a set of procedures to perform some basic manipulations on linked lists of the type shown in Figure 17.7. Specifically, we want to be able to search the list to find a node that has a particular number. We also want a more versatile procedure for adding a node to the list. The procedure `HeadInsert` inserted a node at the head of a list. We will design a procedure that can insert a node at any point in the list, rather than just at the head of the list. Finally, we want a way to delete a node from the list. For example, if the linked list contains a list of student numbers and grades, then one procedure will find a student's record given the student's number. Another procedure will allow us to add a student record. We will also give code for deleting a student record. We will use the same type declarations as those in Figure 17.8.



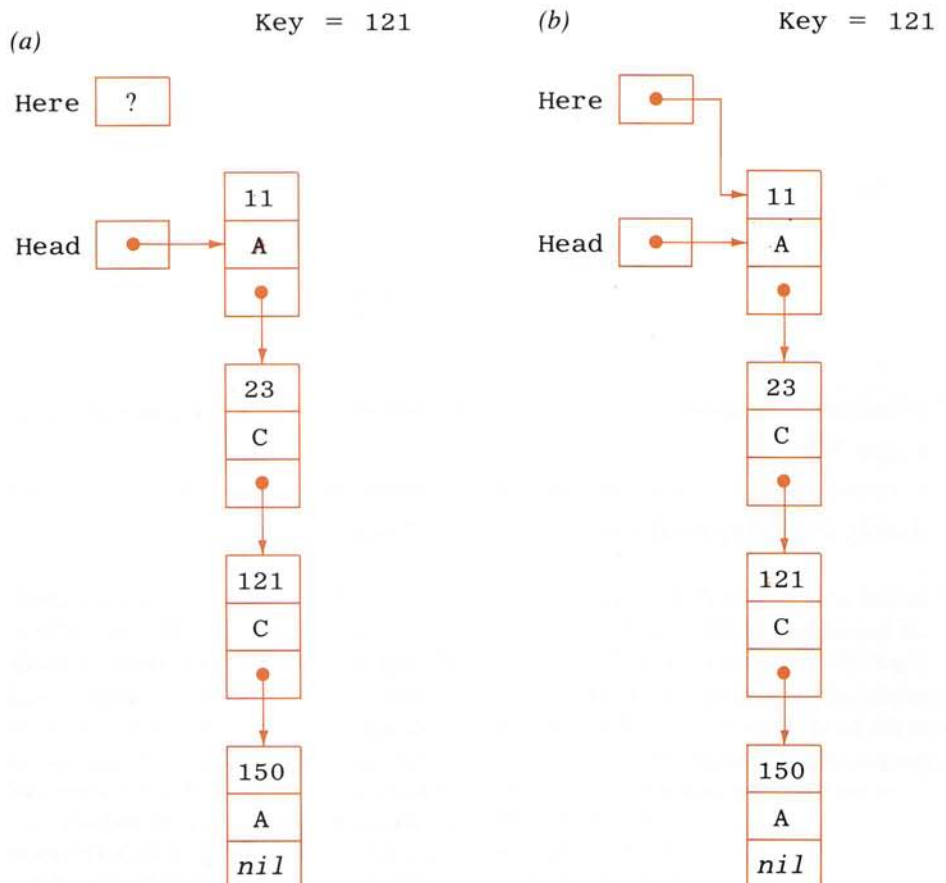
### Problem Definition

searching  
a list

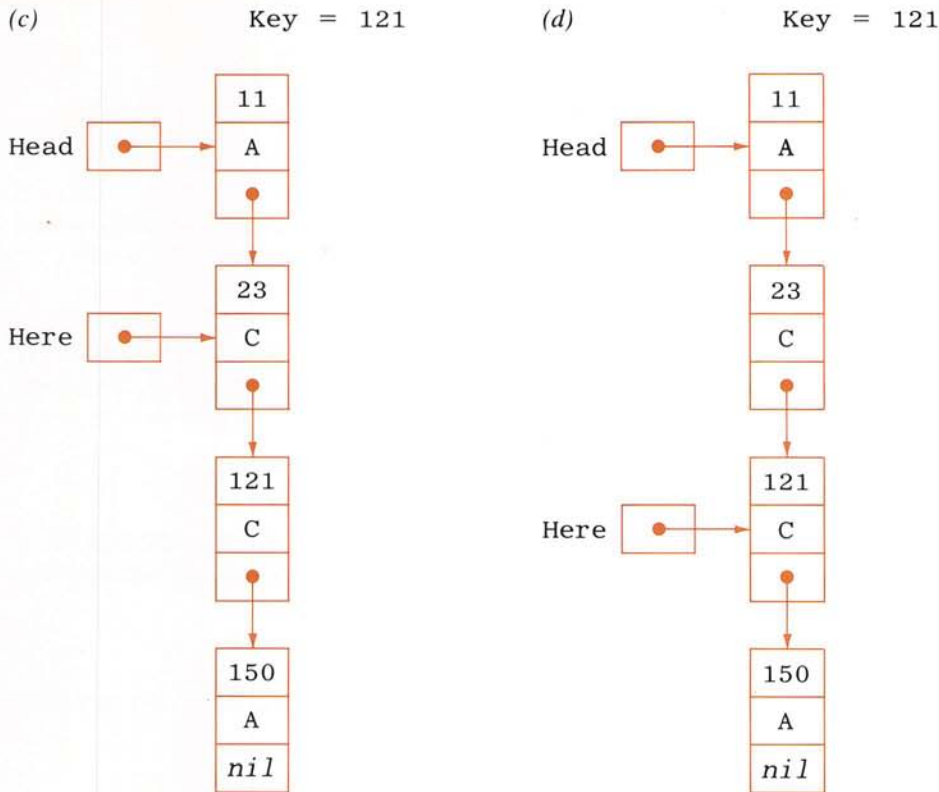
We will first consider the problem of locating a node. We want to design a procedure that will locate a node in a linked list made up of nodes of the type `Node`. More precisely, the procedure has a value parameter `Key` of type `integer` and two variable parameters: `Here` of type `NPointer` and `Found` of type `boolean`. The procedure sets `Found` equal to `true` or `false`, depending on whether or not the list contains a node whose `Number` field has the value `Key`. If there is such a node, then `Here` is left pointing to it.

We assume that the head (first) node is pointed to by a pointer called `Head` and that the end of the list is marked with `nil`. If the list is empty, then the value of `Head` will be `nil`. The pointer `Head` is another parameter to the procedure. The situation is diagrammed in Figure 17.10(a). Our assumptions can be summarized by the following precondition:

*Precondition: Head points to a linked list of nodes of type Node; the end of the list is marked by nil; if the list is empty, then Head = nil.*



**Figure 17.10**  
How the procedure  
Search works.



**Figure 17.10**  
(continued)

The goal for a sample situation is shown in Figure 17.10(d). The goal can be expressed precisely by the postcondition

*Postcondition: If there is a node that contains the integer Key, then Here points to the first such node and Found is equal to true. If no node contains Key, then Found is equal to false.*

## Discussion

The only way to move around a linked list, or any other data structure made up of nodes and pointers, is to follow the arrows. So we will place the pointer Here at the first node and then move it from node to node, following the pointers until we find a node containing the integer Key or until we encounter the end of the linked list. The technique is diagrammed in Figures 17.10(b) through (d). Since empty lists present some minor problems that would clutter our discussion, we will at first assume that the linked list contains at least one node. Later we will come back and make sure the algorithm works for the empty list as well. This search technique yields the following algorithm:

**ALGORITHM**

```

Make Here point to the head (first node) in the list;
while (Here is not pointing to a node containing Key) and
    (Here is not pointing to the last node) do
    make Here point to the next node in the list;
if Here ↑ contains Key, then the number is in the node
    pointed to by Here; otherwise, it is not on the list.

```

We can now translate this algorithm into Pascal code. Since the pointer Head points to the first node, the following will leave Here pointing to the first node:

```
Here := Head
```

In order to move the pointer Here to the next node, we must think in terms of the named pointers we have available. The next node is the one pointed to by the pointer field of the current node pointed to by Here. The node currently pointed to by Here is  $\text{Here} \uparrow$ . The pointer field of that node is

```
Here ↑ . Link
```

In order to move Here to the next node, we want to change Here so that it points to *the node that is pointed to by the above-named pointer field*. Hence, the following will move the pointer Here to the next node in the list:

```
Here := Here ↑ . Link
```

**refinement**

Putting the pieces together yields the following refinement of the algorithm pseudocode:

```

Here := Head;
while (Here ↑ . Number <> Key) and (Here ↑ . Link <> nil) do
    Here := Here ↑ . Link;
if Here ↑ . Number = Key, then the number is in the node pointed to
    by Here; otherwise, it is not on the list.

```

**empty  
list**

We still must go back and take care of the empty list. If we check the above algorithm, we find that there is a problem with the empty list. If the list is empty, then Here is equal to *nil* and hence the following expression is undefined:

```
Here ↑ . Number
```

Hence, we make a special case of the empty string. The complete procedure is given in Figure 17.11.

**Problem Definition****inserting  
nodes**

We next design a procedure to insert a node at a specified place in a linked list. Since we may want the nodes in some particular order, such as in numeric order, we cannot simply insert the node at the beginning (head) of the list nor at the end of the list. We will therefore design the procedure to insert a node between two specified nodes in a linked list. We assume that some other procedure or program part has placed two point-



```

type NPointer = ↑Node;
  Node = record
    Number: integer;
    Grade: char;
    Link: NPointer
  end;

procedure Search(Key: integer; Head: NPointer;
  var Here: NPointer; var Found: boolean);
{Precondition: Head points to a linked list of nodes of type Node; the end
of the list is marked by nil; if the list is empty, then Head = nil.
Postcondition: If there is a node that contains the integer Key, then Here
points to the first such node and Found is equal to true. If no node
contains Key, then Found is equal to false.}
begin{Search}
  if Head = nil then
    Found := false
  else
    begin{nonempty list}
      Here := Head;
      while (Here↑.Number <> Key) and (Here↑.Link <> nil) do
        Here := Here↑.Link;
      {Here is either pointing to a node containing Key or
       is pointing to the last node in the list (or both).}
      Found := (Here↑.Number = Key)
    end {nonempty list}
  end; {Search}

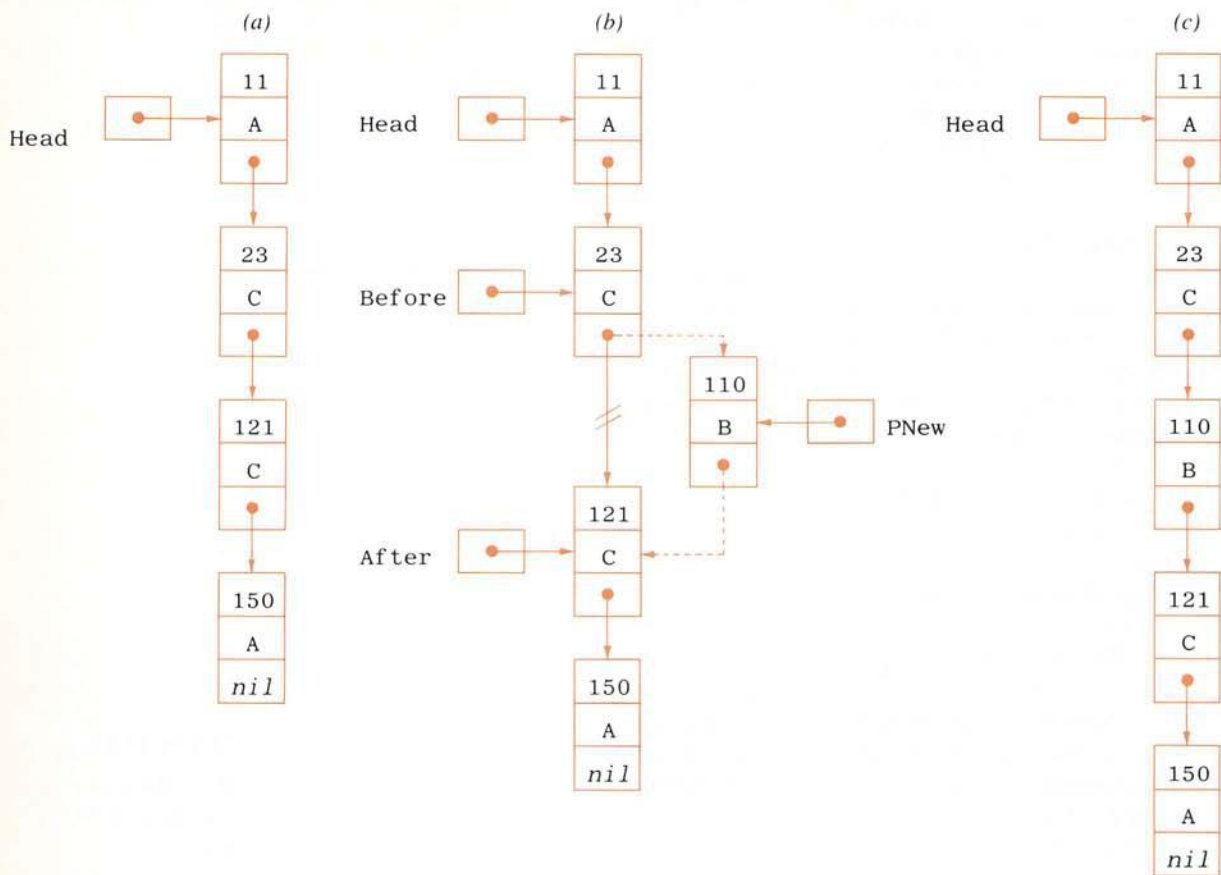
```

**Figure 17.11**  
**Procedure to locate**  
**a node in a linked**  
**list.**

ers called *Before* and *After* pointing to two nodes in the list, as shown in Figure 17.12(b). We wish to insert a new node with new data as shown in Figure 17.12(c). The data for the new node will be of the type *Data* as defined in Figure 17.8. The procedure will create a new node, copy the data into the new node, and insert the node into the linked list. (The pointer *After* is not absolutely necessary, but does simplify our reasoning.)

## Discussion

A new node is set up in the same way that we did in the procedure *HeadInsert*. The difference between this procedure and that one is that we now wish to insert the node not at the head of the list but in some location inside the list. The method of inserting the node is shown in Figure 17.12b. The way to express the indicated resetting of pointers is given below:



**Figure 17.12**  
Inserting a node in  
the middle of a  
linked list.

#### ALGORITHM

Before  $\uparrow$ .Link := PNew; {Redirect a link to the new node}  
 PNew  $\uparrow$ .Link := After {Set a link from the new node back to the list.}

The complete procedure is given in Figure 17.13.

*insertion  
at the ends*

The procedure `Insert` will not work for inserting a node at the beginning or the end of a linked list. The procedure `HeadInsert` in Figure 17.8 can be used to insert a node at the head of the list. Exercise 7 in the next section indicates how the procedure `Insert` can be changed so that it also can insert a node at the end of a list.

```

type NPointer = ↑Node;
  Node = record
    Number: integer;
    Grade: char;
    Link: NPointer
  end;
  Data = record
    Number: integer;
    Grade: char
  end;

procedure Insert(NewData: Data;
                 Before, After: NPointer);
{Inserts a node containing NewData between
 the two nodes pointed to by Before and After.}
var PNew: NPointer;
begin{Insert}
  new (PNew);
  PNew↑.Number := NewData.Number;
  PNew↑.Grade := NewData.Grade;
  Before↑.Link := PNew;
  PNew↑.Link := After
end; {Insert}

```

**Figure 17.13**  
**Procedure to insert**  
**a node in a linked**  
**list.**

By using the procedure `Insert`, we can maintain the linked list in numerical order without rewriting existing nodes. We could “squeeze” a new node into the correct position simply by adjusting two pointers. Furthermore, this is true no matter how long the linked list is or where in the list we want the new record to go. If we had instead used an array of records, then much, and in extreme cases all, of the array would have to be copied over in order to make room for a new record in the correct spot. In spite of the overhead involved in positioning the pointers, inserting into a linked list is frequently more efficient than inserting into an array.

Deleting a node from a linked list is also quite easy. Figure 17.14 illustrates the method. Once the pointers `Before` and `Discard` have been positioned, all that is required to delete the node is the following Pascal statement:

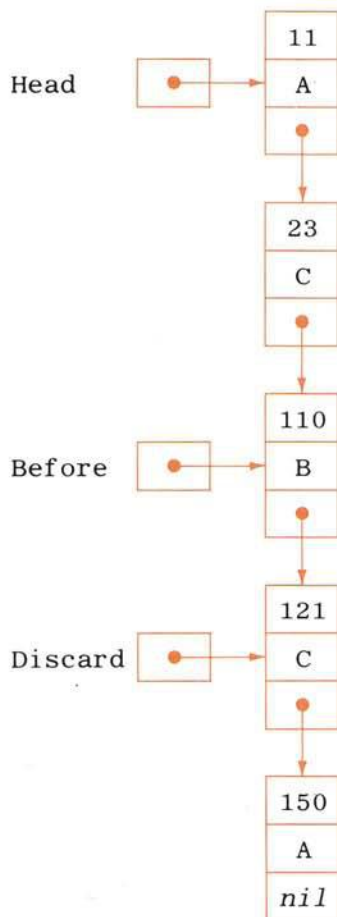
```
Before↑.Link := Discard↑.Link
```

*comparison  
to arrays*

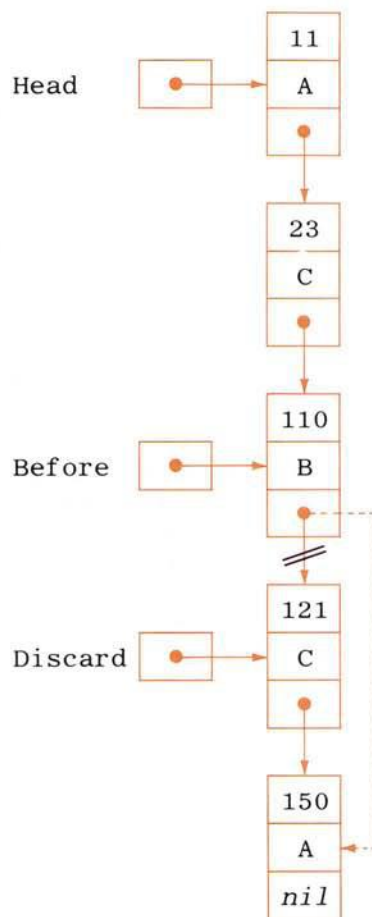
*deleting  
nodes*



(a)

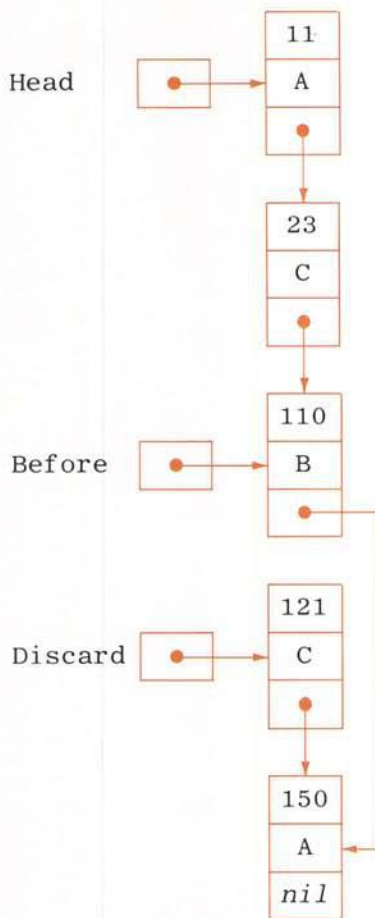


(b)

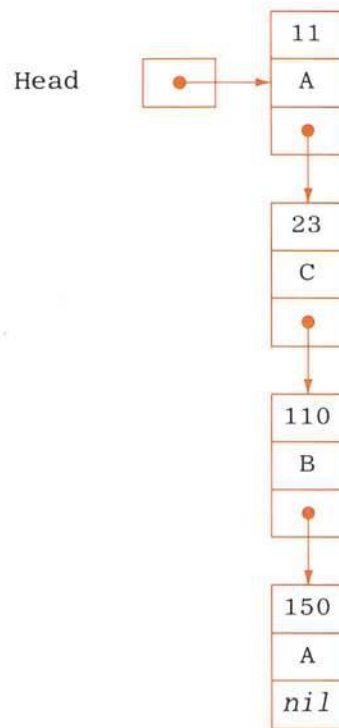


**Figure 17.14**  
Deleting a node  
from a linked list.

(c)



(d)

Figure 17.14  
(continued)

She was poor but she was honest,  
And her parents were the same,  
Till she met a city feller,  
And she lost her honest name.

*Old army song*

## Self-Test Exercises

1. What is the output produced by the following code? All the pointers are of type  $\uparrow$  integer.

```

new (P1);
new (P2);
P1  $\uparrow$  := 10;
P2  $\uparrow$  := 20;
writeln(P1  $\uparrow$ , P2  $\uparrow$ );
P1 := P2;
writeln(P1  $\uparrow$ , P2  $\uparrow$ );
P1  $\uparrow$  := 30;
writeln(P1  $\uparrow$ , P2  $\uparrow$ );
P2  $\uparrow$  := 40;
writeln(P1  $\uparrow$ , P2  $\uparrow$ )

```

2. What is the output produced by the following code? All the pointers are of type  $\uparrow$  integer.

```

new (P1);
new (P2);
P1  $\uparrow$  := 10;
P2  $\uparrow$  := 20;
writeln(P1  $\uparrow$ , P2  $\uparrow$ );
P1  $\uparrow$  := P2  $\uparrow$ ;
writeln(P1  $\uparrow$ , P2  $\uparrow$ );
P1  $\uparrow$  := 30;
writeln(P1  $\uparrow$ , P2  $\uparrow$ );
P2  $\uparrow$  := 40;
writeln(P1  $\uparrow$ , P2  $\uparrow$ )

```

3. How would the output of the program in Figure 17.3 change if the lines

```

P2  $\uparrow$ .Number := 3;
P2  $\uparrow$ .Grade := 'C';

```

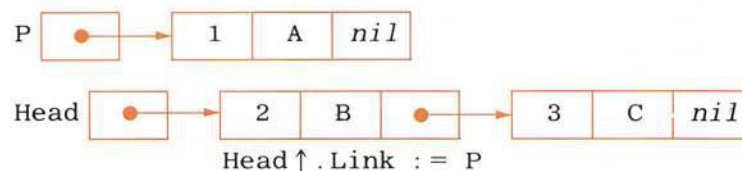
were replaced by the following lines?

```

P1  $\uparrow$ .Number := 3;
P1  $\uparrow$ .Grade := 'C';

```

4. Given the situation diagrammed below, what is the effect of the assignment statement shown? The pointers and nodes are of the types given in Figure 17.13.





5. Write a procedure to fill a linked list with the integers 1 through N. N should be a parameter.
6. Write a procedure to display to the screen all the integers in a linked list of the same general kind as you used in the previous exercise. (The list may be of a different length and may contain numbers other than the ones that would be placed there by the procedure of the last exercise.)
7. The pointer parameter *After* in the procedure *Insert* in Figure 17.13 is not absolutely necessary. Rewrite the procedure so that it works without this parameter.

## Pointer Functions

In Pascal, functions may return pointers. In Figure 17.11 we designed a procedure to search a linked list and to set the value of a variable parameter equal to a pointer that is pointing to the desired record. Alternatively, we could design a function to return the pointer. The function heading might be as follows:

```
function Find(Key: integer; Head: NPointer): NPointer;
{ Returns a pointer that points to the first node containing Key;
  returns nil if Key is not in the list.
  Precondition: Head points to a linked list of nodes of type Node; the end
  of the list is marked by nil; if the list is empty, then Head = nil. }
```

Since a function can return only one type of value, we have used the pointer *nil* to indicate that the desired node was not found on the list.

We could design the body of the function *Find* to be similar to the body of the procedure *Search* in Figure 17.11. However, we will instead use a recursive algorithm in the design of this function. Linked lists and other data structures made using pointers lend themselves to recursive algorithms because the structure is repeated. The pointer *Head* points to a linked list. The pointer coming out of the first node (*Head* ↑ . *Link*) points to another, shorter linked list, namely the one starting with the second node of the longer list. This repeated structure allows us to state very simple recursive algorithms.

```
if Head = nil then
  Key is not on the list
else if Head ↑ . Number = Key then
  Key is in the node pointed to by Head
else
  Search the linked list pointed to by Head ↑ . Link
```

*recursion  
and  
pointers*

*recursive  
ALGORITHM*

A complete function declaration is displayed in Figure 17.15.

```

type NPointer = ↑ Node;
Node = record
    Number: integer;
    Grade: char;
    Link: NPointer
end;

function Find(Key: integer; Head: NPointer): NPointer;
{ Returns a pointer pointing to the first node containing Key; returns nil if Key is not in the list.
Precondition: Head points to a linked list of nodes of type Node; the end
of the list is marked by nil; if the list is empty, then Head = nil. }
begin{Find}
    if Head = nil then
        Find := nil
    else if Head↑.Number = Key then
        Find := Head
    else
        Find := Find(Key, Head↑.Link)
    end; {Find}

```

**Figure 17.15**  
Function that  
returns a pointer.

## Pitfall

### Testing for the Empty List

When designing recursive algorithms, the stopping case is typically the empty list, which is represented by the pointer *nil*. Hence, even if you know that the linked list or other structure in your program will not be empty, you may still need to include a test for the empty list. For example, the function *Find* in Figure 17.15 can terminate when the value of the parameter *Head* is equal to *nil*, and so a series of recursive calls starting with a nonempty list can end with this case.

When dealing with the pointer *nil*, remember that it does not point to anything, and so any reference to a node that it points to is undefined. For example, again consider the function *Find*. The test for *nil* must be the first case in the nested *if-then-else* statement. The reason for this is that if the value of *Head* is *nil*, then *Head*↑.Number is undefined.

## Case Study

### Using a Linked List to Sort a File

#### Problem Definition

If we wish to sort a file of records, then we can read the records into some other location as we sort them. A linked list is a convenient intermediate data structure to use for this purpose. As an example, we will design a program to sort a file of records of the type declared below:

```
type Data = record
    Number: integer;
    Grade: char
end;
```

We assume that the file variable is `GradeFile` and that the records might be in any order. We want the program to change the file so that it contains the same records but so that they are sorted from smallest to largest according to the `Number` field.

#### Discussion

One possible way to proceed is to copy the data into a linked list inserting each piece of data in the correct place so as to keep the list sorted. After the linked list is constructed the program can simply copy the sorted list back into the file. This method works in both standard Pascal and TURBO Pascal. However, the details for opening and naming files, which we use in the final program, will only work for TURBO Pascal. (With slight modifications the program can be made to work in standard Pascal.)

One breakdown of the task into subtasks yields the following algorithm outline:

ALGORITHM

- I. `BuildList`: for each record in the file:
  1. read the record
  2. `FindSlot`: find the correct place to insert the record data in the list.
  3. `Insert`: insert a new node with the record data in the list.
- II. `CopyToFile`: copy the linked list back into the file.

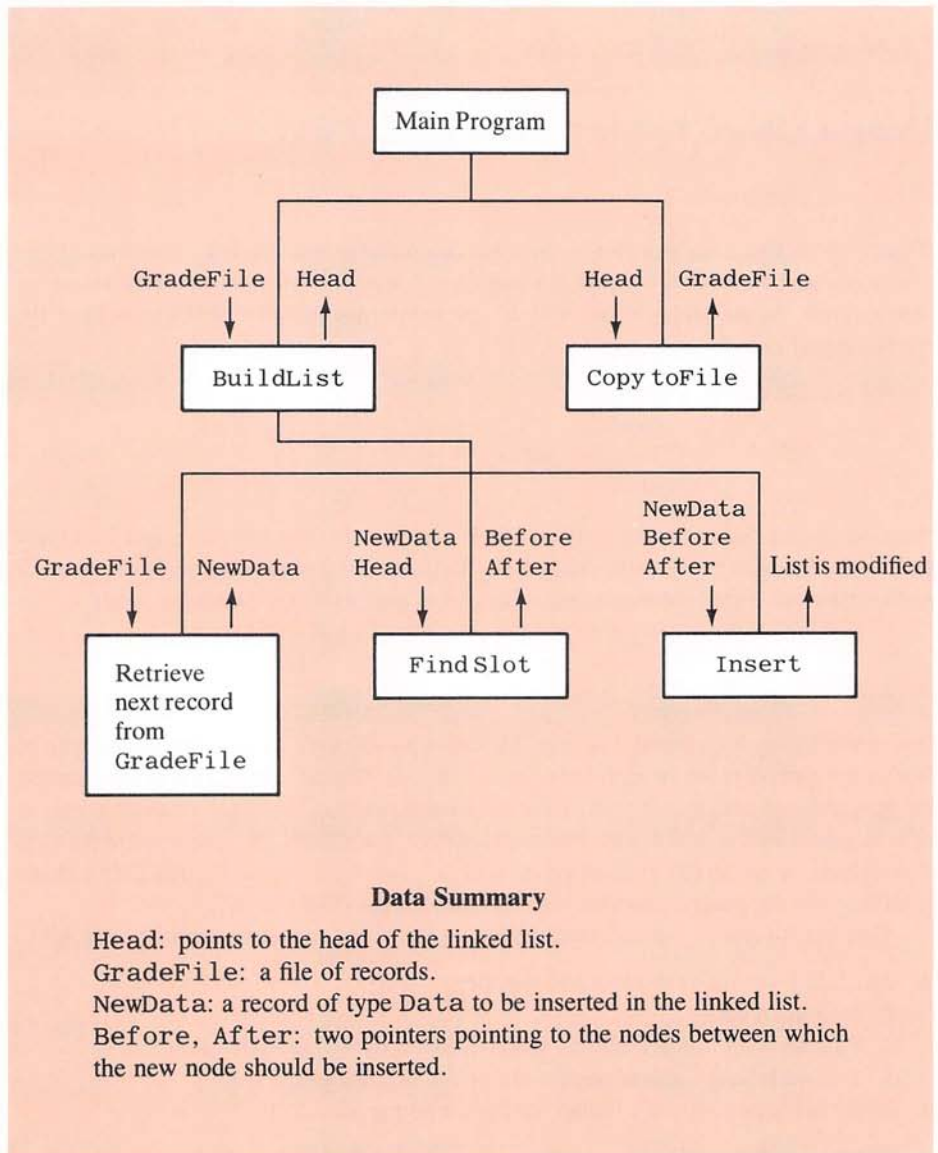
Figure 17.16 contains the data flow diagram for the algorithm. For the procedure `Insert` we can use the procedure by that name in Figure 17.13. The procedure `FindSlot` positions the two pointers `Before` and `After` so that they point to two adjacent nodes such that the new node belongs between these two nodes. Expressed more precisely, this means that the following must hold after the procedure `FindSlot` has completed:

$\text{Before} \uparrow . \text{Number} \leq (\text{the number in the node to be inserted}) \leq \text{After} \uparrow . \text{Number}$

If the list is empty, as it is at the start, or if the node to be inserted belongs at the start or end of the list, then there will not be two nodes with these properties. Rather

*sentinel  
nodes*





**Figure 17.16**  
 Data flow diagram  
 for building a  
 sorted linked list.

than make special cases for empty lists and insertions at the ends of a list, we will instead add two sentinel nodes to mark the two ends of the list. The first sentinel node will contain a number smaller than all the numbers in the file. The other sentinel node will contain a number larger than all the nodes in the file. Since the list will initially contain these two nodes, the case of the empty list is eliminated. Since these two nodes will have numbers that ensure that they stay on the ends of the list, the case of inserting

a node at the end of the list is also eliminated. The procedure FindSlot can thus be written without any special cases. The complete program, including the procedure FindSlot, is given in Figure 17.17.

```

program SortFile; {TURBO Pascal version}
{Sorts the records in GRADE.DAT by their Number fields. Works by reading
records into a linked list, sorting them by insertion in the process; the records
are then read back into the file, leaving a sorted file. Assumes that the
constants Small and Large have the properties stated in their comments.}
const Small = -maxint; {A number smaller than any number in the file.}
      Large = maxint; {A number larger than any number in the file.}
type NPointer = ↑Node;
      Node = record
          Number: integer;
          Grade: char;
          Link: NPointer
      end;
      Data = record
          Number: integer;
          Grade: char
      end;
      RecordFile = file of Data;
var Head: NPointer;
      GradeFile: RecordFile;

procedure FindSlot(NewData: Data; Head: NPointer;
                  var Before, After: NPointer);
{Precondition: Head points to the head of a linked list; the Number fields
of the nodes are in increasing order starting from the head. The ends of the
list are marked with two sentinel nodes, one containing the number Small
and one containing the number Large.
Postcondition: Before and After point to nodes of the list and
Before ↑ .Number <= NewData.Number <= After ↑ .Number.}
begin{FindSlot}
    Before := Head;
    After := Head ↑ .Link;
    while (Before ↑ .Number > NewData.Number) or
          (NewData.Number > After ↑ .Number) do
        begin{move pointers one node}
            Before := Before ↑ .Link;
            After := After ↑ .Link
        end {move pointers one node}
    end; {FindSlot}

```

**Figure 17.17**  
Sorting with a  
linked list.

```

procedure Insert(NewData: Data;
                 Before, After: NPointer);
{Inserts a node containing NewData between
the two nodes pointed to by Before and After.}
var PNew: NPointer;
begin{Insert}
  new(PNew);
  PNew↑.Number := NewData.Number;
  PNew↑.Grade := NewData.Grade;
  Before↑.Link := PNew;
  PNew↑.Link := After
end; {Insert}

procedure BuildList(var GradeFile: RecordFile; var Head: NPointer);
{Creates a linked list of records from records in the file GradeFile. The linked
list is sorted by the Number field. Head is left pointing to the head of the list;
An extra node containing the number Small is at the head of the list and
an extra node containing the number Large is at the end of the list.}
var Before, After: NPointer;
    NewData: Data;
begin{BuildList}
  new(Head);
  Head↑.Number := Small;
  new(Head↑.Link);
  Head↑.Link↑.Number := Large;
  {The linked list has two nodes; the first with a number smaller
than any record; the last with a number larger than any record.}
  reset(GradeFile);

  while not eof(GradeFile) do
    begin{one record}
      read(GradeFile, NewData);
      FindSlot(NewData, Head, Before, After);
      Insert(NewData, Before, After)
    end {one record}
  end; {BuildList}

procedure CopyToFile(Head: NPointer; var GradeFile: RecordFile);
{Copies the data in all nodes except the first and last nodes of a linked list
into the file GradeFile. Precondition: Head points to the head of the linked list;
the first node contains the number Small; the last node contains the number Large.}
var Finger: NPointer;
    OneRecord: Data;

```

**Figure 17.17**  
(continued)



```

begin{CopyToFile}
  rewrite(GradeFile);
  Finger := Head↑.Link; {Pass over node with Small.}
  while Finger↑.Number <> Large do
    begin{one record}
      OneRecord.Number := Finger↑.Number;
      OneRecord.Grade := Finger↑.Grade;
      write(GradeFile, OneRecord);
      Finger := Finger↑.Link
    end {one record}
  end; {CopyToFile}

begin{Program}
  assign(GradeFile, 'GRADE.DAT');
  writeln('Reading GradeFile into a sorted list. ');
  BuildList(GradeFile, Head);
  writeln('Copying sorted list into GradeFile. ');
  CopyToFile(Head, GradeFile);
  close(GradeFile);
  writeln('GradeFile sorted. ')
end. {Program}

```

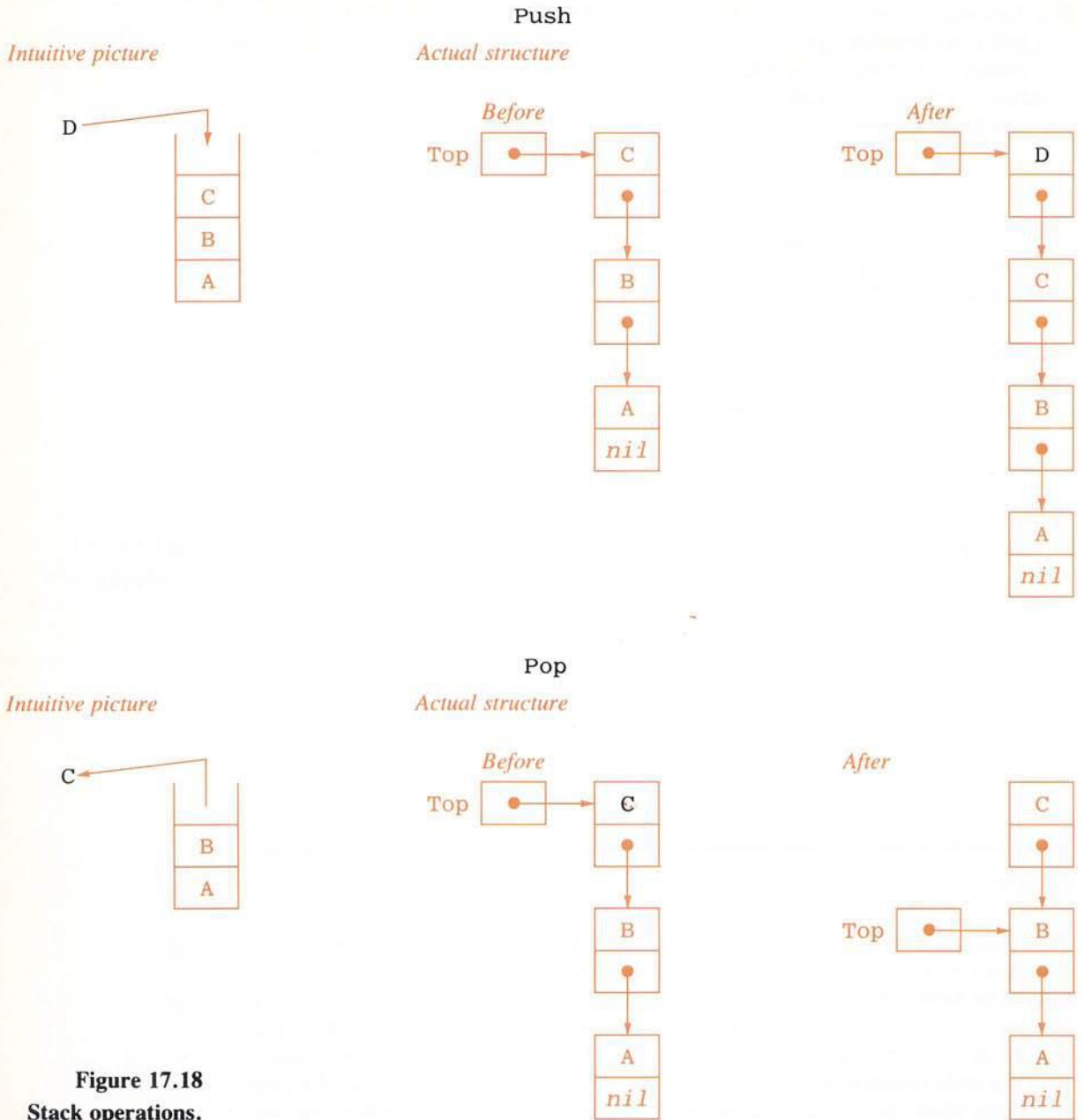
**Figure 17.17**  
(continued)

---

## Stacks

We introduced the notion of a stack in Chapter 14 and showed how it could be used to keep track of recursive procedure calls and local variables. One way to implement a stack is by means of a linked list. In this implementation a stack is nothing but a linked list that is used in a restricted way, namely, nodes are only added at the head of the list and are only deleted from the head of the list. When you are dealing with stacks, adding a node is usually called a *push* operation, and deleting a node is called a *pop* operation. These operations and their implementations as operations on linked lists are illustrated in Figure 17.18. This terminology will explain the procedure names used in Figure 17.19. That program uses a stack of characters to read in a line of text and output it in reverse order. The procedure *Push* is essentially the same as the procedure *HeadInsert* we designed earlier, except that the nodes have one less data field. The procedure *Pop* simply moves the pointer at the head of the list in the manner shown in Figure 17.18.

*push*  
*and*  
*pop*



**Figure 17.18**  
Stack operations.

**Program**

```

program Reverse(input, output);
{Reads a line of text and outputs it written backwards.}

type FPointer = ↑Frame;
   Frame = record
       Data: char;
       Link: FPointer
   end;
var Top: FPointer; {Points to the top of the stack.}
    Symbol: char;

procedure Push(NewData: char; var Top: FPointer);
{Pushes a frame (node) containing the data onto the stack.
Top points to the top frame (head node) on the stack.}
var PNew: FPointer;
begin{Push}
    new (PNew); {Creates a new node to hold NewData.}
    PNew↑.Data := NewData;
    PNew↑.Link := Top;
    {Places the new node at the head of the list.}
    Top := PNew
    {Moves Top so it points to the new top (head) of the stack (list).}
end; {Push}

function Empty(Top: FPointer): boolean;
{Returns true if the stack pointed to by Top is empty.}
begin{Empty}
    Empty := (Top = nil)
end; {Empty}

procedure Pop(var Top: FPointer; var Out: char);
{Pops one frame(node) off the stack pointed to by Top;
Out is set equal to the Data field of the popped frame (node).}
begin{Pop}
    if Empty(Top) then
        writeln('Error: attempted to pop an empty stack.')
    else
        begin{Popping}
            Out := Top↑.Data;
            Top := Top↑.Link
        end {Popping}
    end; {Pop}

```

**Figure 17.19**  
**Program using a**  
**stack.**



```

begin{Program}
  writeln('Enter a line of text. ');

  Top := nil; {Makes an empty stack.}

  while not eoln do
    begin{reading a symbol}
      read(Symbol);
      Push(Symbol, Top)
    end; {reading a symbol}
  readln;

  while not Empty(Top) do
    begin{writing a symbol}
      Pop(Top, Symbol);
      write(Symbol)
    end; {writing a symbol}
  writeln;

  writeln('That''s it, forwards and backwards!')
end. {Program}

```

#### Sample Dialogue

```

Enter a line of text.
Able was I ere I saw Elba. Cute?
?etuC .able was I ere I saw elbA
That's it, forwards and backwards!

```

Figure 17.19  
(continued)

---

### dispose (Optional)

Look again at Figure 17.18. The node that is popped off the stack is no longer on the stack (linked list) and it cannot be referenced by the program. It is a lost node, but it has not been destroyed. Unless the program explicitly eliminates the node, it will remain in storage and will waste storage. The predefined procedure `dispose` can be used to eliminate useless dynamic variables and so free some storage for other purposes. For example, the following will eliminate the dynamic variable (node) pointed to by `Discard`:

```
dispose(Discard)
```

In Figure 17.20, we have rewritten the procedure `Pop` from Figure 17.19 so that it uses `dispose` and so does not waste storage. For the program in Figure 17.19, it makes little difference which version of `Pop` we use. However, if we were to enclose the body of the program in a loop that repeated the reading and writing a large number

---

```

procedure Pop(var Top: FPointer; var Out: char);
{Pops one frame(node) off the stack pointed to by Top;
 Out is set equal to the Data field of the popped frame (node).}
var Discard: FPointer;
begin{Pop}
  if Empty(Top) then
    writeln('Error: attempted to pop an empty stack.')
  else
    begin{Popping}
      Out := Top↑.Data;
      Discard := Top; {Discard points to the node being popped.}
      Top := Top↑.Link;
      dispose(Discard)
    end {Popping}
end; {Pop}

```

**Figure 17.20**  
(Optional)  
Procedure using  
dispose.

of times, then the one without dispose would generate extra lost nodes on every iteration of the loop. These lost nodes would remain in storage and so consume storage. If the loop iterates enough times, the lost nodes could even cause the program to terminate because of lack of storage. The version with dispose will need only enough storage for one line of text.

In TURBO Pascal there is an alternative to dispose. In TURBO Pascal the predefined procedures mark and release may be used instead. (mark and release are not implemented in standard Pascal.) Before we describe these two TURBO Pascal procedures, we need to explain how pointers are implemented in memory. We do this in the next section.

---

## Implementation (Optional)

In order to program in a high level language such as Pascal, you do not need to know how the language is implemented, any more than you need to understand the workings of the human larynx or of the human brain in order to use the English language. Pascal programs are implemented as the machine code produced by the Pascal compiler, and all you need to know is that the machine code makes the input and output behave as we have described for the language Pascal. Still, it is sometimes helpful, and invariably interesting, to know some details of the implementation. This is particularly true of pointers and dynamic variables. Since the description of their implementation is very much more concrete than the high level description of Pascal pointers, many people find it easier to understand pointers in terms of their implementation. Additionally, it gives you a good idea of how the notion of pointers can be implemented in other programming languages, including many high level programming languages that do not have pointers as a basic predefined construction.

---

address

In order to describe a typical implementation for Pascal pointers and dynamic variables, we need to recall our discussion of the internal structure of computers. Recall that a computer's main memory consists of a very long sequence of numbered memory locations, and that each memory location can hold one string of binary digits, which we can interpret as a data value of some simple type, such as an integer or a character. The locations are frequently called *words*, and the number of a location is frequently called its *address*. As this discussion indicates, a computer memory is structured much like a very large one-dimensional array.

To be concrete, let us say that we want to implement a linked list of the type shown in Figure 17.12(a). Each node will contain an integer field and a field of type *char* in addition to the one pointer field. One way to do this is to allocate three adjacent memory locations for each dynamic variable that is to serve as a node in the linked list. One of the three locations will hold the integer in the node, one will hold the character, and the third location will hold some integer that can be interpreted as a pointer to a node.

What can be interpreted as a pointer to one of these dynamic variables? These dynamic variables are implemented as three adjacent memory locations. Hence, one way to name one of these dynamic variables is to name the three addresses of these locations. That is exactly what we will do; however, we will use only the first address, since the other two are trivial to compute from the first one. In our implementation a dynamic variable of the type under discussion is just three adjacent memory locations: the first two hold the integer and character values, and the third holds the pointer. In this implementation the pointer is realized as the address of the first of the three memory locations that represent the dynamic variable that is pointed to.

By way of example, consider Figure 17.21. It shows a possible implementation of the linked list shown in Figure 17.12(a). The *nil* pointer is indicated by the number minus one. Minus one is used for *nil* because we know there is no location with that address. Any other negative number would do as well. The right-hand figure is an abstraction that ignores the particular address numbers used. Since the particular address numbers used are not important to the realization, that right-hand figure is easier to deal with.

adding  
nodes

As a program proceeds to add and delete nodes from a linked list, the picture of memory becomes a good deal more intricate. Suppose we wish to add a node containing the integer 110 and the character 'B' in a position that will keep the nodes in the linked list in numeric order; the result should be as shown in Figure 17.12(c). Figure 17.22 shows the configuration of memory after the dynamic variable has been added. Note that the dynamic variable for this node was implemented with the next three available memory locations. The way to think about a linked list is as if the new node is squeezed in. However, in this implementation all the old nodes stay where they are. Only the values of the pointer fields change.

deleting  
nodes

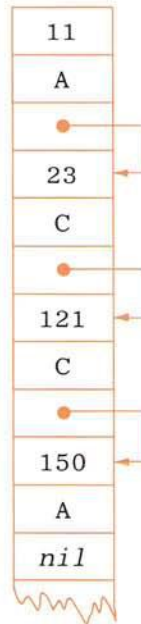
Figure 17.23 shows the implementation of the same linked list after adding the node containing 110 and 'B', as just described, and then deleting the node containing 121 and 'C'.

garbage  
collection

Notice that as we add nodes, the pattern of arrows gets to be rather messy, but that need not concern us. Ordinarily, we need not be concerned with the actual memory addresses when we use pointers and dynamic variables. We need only think in terms of an abstraction of the pointer structure that ignores the actual locations of the dynamic variables. If we have enough memory, we can get away with thinking only on an ab-

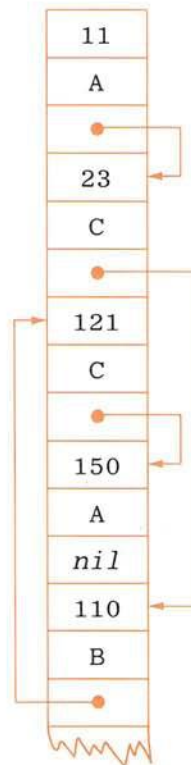


1000	11
1001	A
1002	1003
1003	23
1004	C
1005	1006
1006	121
1007	C
1008	1009
1009	150
1010	A
1011	-1



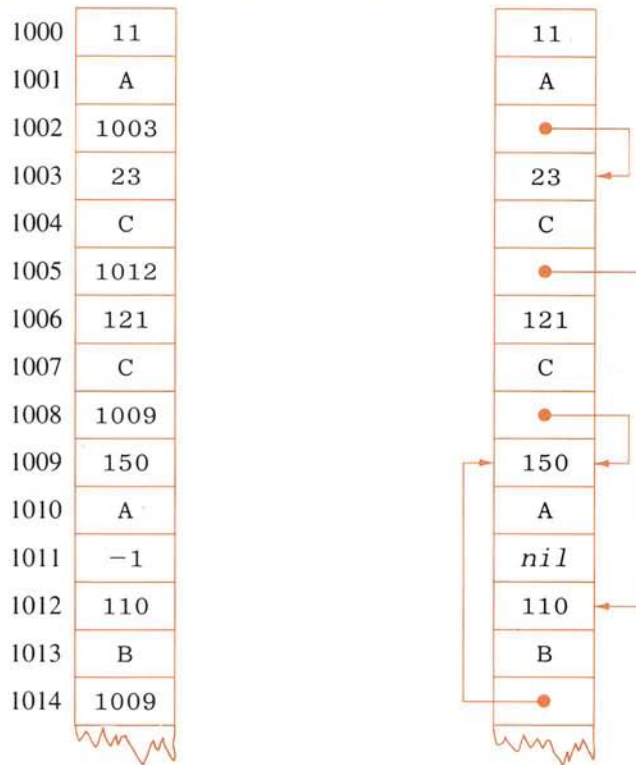
**Figure 17.21**  
(Optional)  
**Implementation**  
of a linked list.

1000	11
1001	A
1002	1003
1003	23
1004	C
1005	1012
1006	121
1007	C
1008	1009
1009	150
1010	A
1011	-1
1012	110
1013	B
1014	1006



**Figure 17.22**  
(Optional)  
**Implementation of**  
a linked list with  
added node.

**Figure 17.23**  
(Optional)  
**Implementation of**  
**a linked list with**  
**deleted node.**



abstract level, which ignores all the details of the particular memory addresses used. Unfortunately, there is some danger of wasting memory if we think exclusively on this abstract level. Look again at the memory configuration shown in Figure 17.23. Notice that the dynamic variable in locations 1006, 1007, and 1008 still has an integer, a letter, and a pointer in it. Yet the node represented by that dynamic variable is no longer on the linked list. The program will never be able to use the dynamic variable stored in locations 1006 through 1008, and so those memory locations should be made available for other uses. Yet if we continue to add dynamic variables at the bottom of memory, then locations 1006 through 1008 will never be reused. Locations like 1006, 1007, and 1008 are frequently referred to by the technical term *garbage*—not a very dignified word, but a descriptive one and the one that is generally used. A good implementation would keep track of these garbage memory locations and reuse them. Locating such garbage memory locations so that they can be reused is called, appropriately enough, *garbage collection*.

Many implementations of Pascal do not have very good garbage collection, and the system must be given some help in order to perform this task. Specifically, the system must be told which dynamic variables are garbage. This is what the *dispose* command does.

The idea of this implementation can be used in high level languages as well. If

Pascal did not have pointers as a built-in feature, we could still implement something like pointers by using an array of records in the same way that we used the computer's main memory in the implementation just described.

## TURBO Pascal

### mark and release

(Optional)

In some versions of TURBO Pascal, `dispose` is not implemented. However, in any TURBO Pascal system, the predefined procedures `mark` and `release` can be used to eliminate dynamic variables. In any one TURBO Pascal program, you may use either `dispose` or `mark` and `release`. However, you may not use both methods in the same program. The procedures `mark` and `release` are not available in standard Pascal.

The procedures `mark` and `release` are best described in terms of the implementation of pointers and dynamic variables that was discussed in the previous section. Recall that the dynamic variables we were considering were records with three fields, an integer, a character, and a pointer field. Such dynamic variables might be used as nodes in a linked list, but their specific use is not important. Each dynamic variable was implemented as three adjacent memory locations, one containing an integer, one containing a character, and one containing a pointer. The pointer was implemented by storing the address of the dynamic variable that it pointed to. A similar implementation is used for dynamic variables of other types. This particular type was used just to have a concrete example.

These implementations have no automatic garbage collection, and so the amount of memory occupied by dynamic variables will grow larger and larger as more dynamic variables are created. The memory occupied by these dynamic variables increases in a very orderly way. When a new dynamic variable is added, it is placed in the next available memory location. That is, it is always added at the end of the list.

The memory allocated to dynamic variables is used as a list, with new dynamic variables added to the end of the list. Another way to think of this memory is to consider it to be a stack, with the end of the list serving as the top of the stack. The command `mark` can be used to mark a position in the stack. The command `release` can be used to release all the memory up to a marked location. The value of any pointer variable that points to a dynamic variable in the released memory space becomes undefined, and all the memory locations in the released memory space are made available for reuse.

`mark` and `release` are only approximately equivalent to `dispose`. They cannot be used to release one particular dynamic variable, unless it just happens to be the last one that was created. They always have to be used on a whole chunk of memory, a whole collection of dynamic variables. In this sense `mark` and `release` are not as versatile as `dispose`. However, `mark` and `release` are more powerful since they can be used to release dynamic variables that have no pointer pointing to them.

*marking  
the stack*



In order to use `mark` and `release`, the program uses a pointer variable of type `↑ integer`. The pointer is not used to point to dynamic variables; instead it is dedicated to the marking process. For example, suppose that `M1` is of type `↑ integer` and consider the following statement:

```
mark (M1)
```

The effect of this statement is to mark a position in memory by having `M1` “point to it.” (We include the quotation marks because we are using the informal notion of “pointing,” not the Pascal notion of pointing.) Later the program can release the memory occupied by all the dynamic variables that have been added in memory locations after the position marked by `M1`. To release that memory, the program executes

```
release (M1)
```

This call to `release` releases all the memory between the position marked by `M1` and the next memory location available for dynamic variables. The value of a pointer variable used for marking memory, such as `M1`, must not be changed between the time it is used with `mark` and the time it is used with `release`.

By using several different pointer variables, in the same way as `M1`, you can mark several different positions in memory and later release all the memory up to any of these marked positions.

Figure 17.24 shows a sample procedure that uses `mark` and `release`. It reads a line of text into a linked list and then writes the line to the screen in reverse order. After that, all the memory used for the linked list is released. Because the procedure uses `mark` and `release` to manage garbage collection, it can be called many times and yet will require only enough storage to hold the longest line of text. Without such garbage collection, each successive call to the procedure would consume more and more memory to hold all the garbage nodes that are no longer needed.

(The procedure `Reverse` in Figure 17.24 could also be written using the `push` and `pop` procedures that are shown in Figure 17.19. In fact, those procedures would normally be preferable. However, we did not do so because we wanted to keep the example as simple as possible.)

---

## Doubly Linked Lists—A Variation on Simple Linked Lists

A variant of the linked list is the doubly linked list such as the one shown in Figure 17.25. A doubly linked list is like a simple linked list except that in a doubly linked list, there are pointers pointing backward as well as forward. Doubly linked lists illustrate the point that a node may contain more than one pointer. Doubly linked lists are handled very much like ordinary linked lists except that they allow the program to move along the list in either direction. This can sometimes be useful. Our handling of linked lists would have been easier if they were doubly linked. In that case we would usually not need trailer pointers such as the pointer `Before` in Figure 17.14. Doubly linked lists do, however, require more storage than ordinary linked lists, because of the extra pointer in each node. A possible set of type declarations for a doubly linked list are as follows:

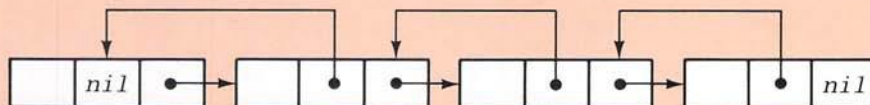
---

```

procedure Reverse;
{TURBO Pascal procedure to read in a line of text
and write it out to the screen in reverse order.}
type ItemPointer = ↑Item;
   Item = record
       Letter: char;
       Link: ItemPointer
   end;
var Head, Next: ItemPointer;
    Spot: ↑integer;
begin{Reverse}
    mark(Spot);
    Head := nil;
    writeln('Enter a line of text:');
    while not eoln do
        begin{while}
            new(Next);
            read(Next↑.Letter);
            Next↑.Link := Head;
            Head := Next
        end; {while}
    readln;
    writeln('Spelled backwards it reads:');
    while Head <> nil do
        begin{Second while}
            write(Head↑.Letter);
            Head := Head↑.Link
        end; {Second while}
    writeln;
    release(Spot)
end; {Reverse}

```

**Figure 17.24**  
TURBO Pascal  
procedure using  
mark and  
release.



**Figure 17.25**  
A doubly  
linked list.

```

type Link = ↑Node;
  Info = record
    Number: integer;
    Price: real;
    Style: 1 . . 8
  end;
Node = record
  Data: Info;
  Back, Forward: Link
end;

```

Notice that in the preceding type declarations, nodes have three fields, one of which is itself a record. A node can be almost any sort of record. A hierarchical arrangement, such as that shown above, is often convenient.

---

## Trees

### binary trees

A useful kind of data structure that is significantly different from the linked list structure is the *binary tree*. A sample binary tree, together with possible type declarations for the pointers and nodes of the tree, is shown in Figure 17.26. That tree stores names and hours worked for the employees of a small firm. (Any reasonable type definition for `CharString` would be acceptable, such as an array of characters or the more complex record type defined in Chapter 11.) To understand why structures such as this are called “trees,” turn the page upside down. The resulting branching structure should, with a little bit of help from your imagination, look like the branching structure of a tree.

In order to simplify the notation in our discussion of trees, we will use a simpler node that stores only a single integer plus the two pointers. However, the techniques we develop will apply to other node types as well. The type declarations we will use are as follows, and a sample tree with nodes of this type is shown in Figure 17.27.

```

type Pointer = ↑TreeNode;
  TreeNode = record
    Data: integer;
    Left, Right: Pointer
  end;
var Root: Pointer;

```

### root node

The pointers called `Root` in Figures 17.26 and 17.27 each point to a special node called the *root node*. The name comes from the fact that if you turn the picture upside down, then that node is located where the root of the tree would start. The root node is the only node from which every other node can be reached by following the pointers. It serves a function similar to that of the head node in a linked list.

### traversing a tree

If we want to list the data in a tree, we must design our program to traverse the nodes in some order and to write out the contents of each node. Algorithms for traversing a tree are expressed most easily in their recursive form.



```

type Branch = ↑NodeRecord;
NodeRecord = record
    Name: CharString;
    Hours: real;
    Left, Right: Branch
end;

var Root: Branch;

```

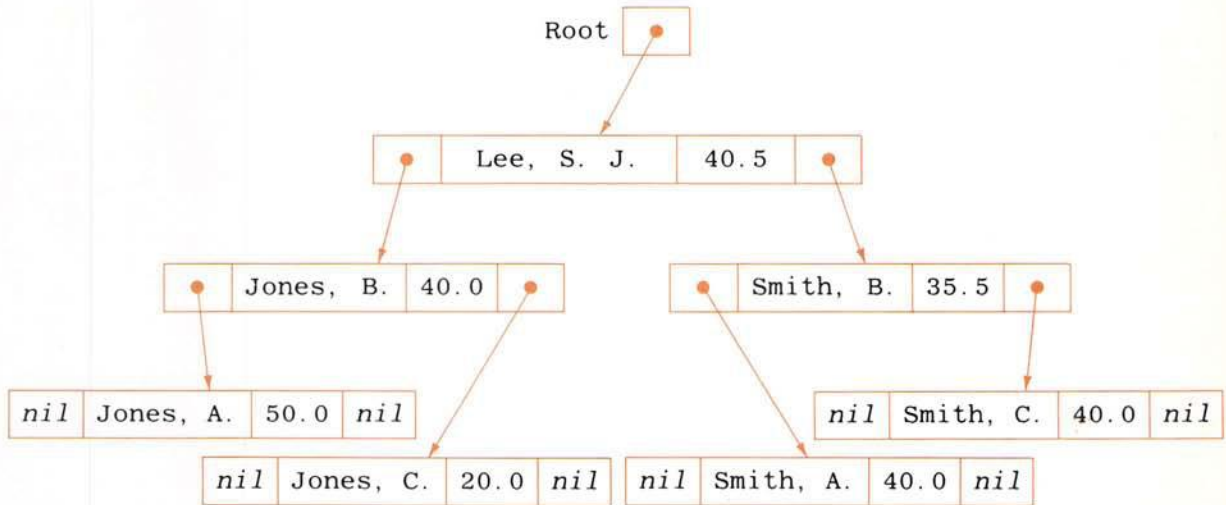


Figure 17.26  
A binary tree.

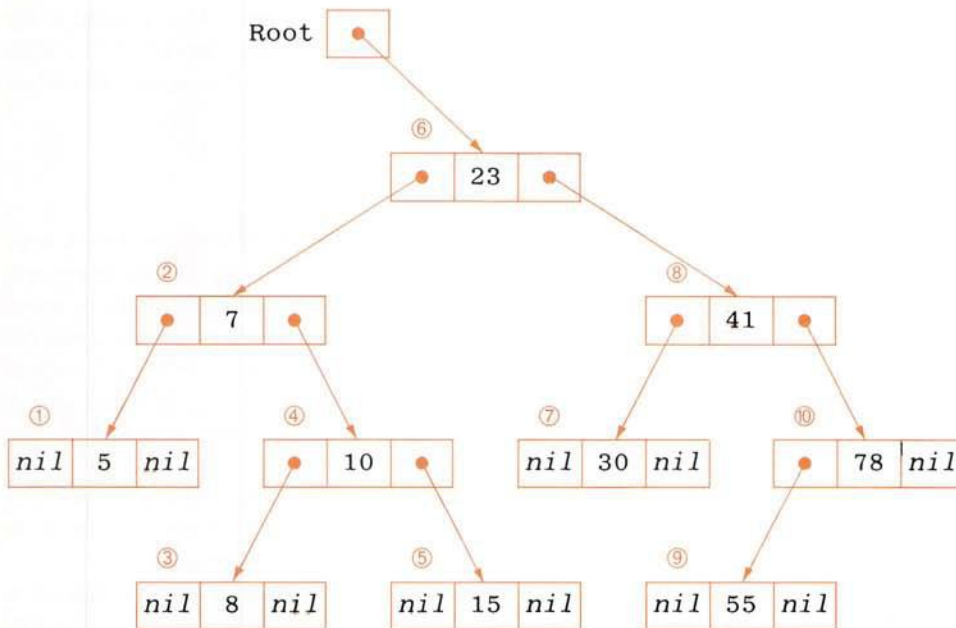
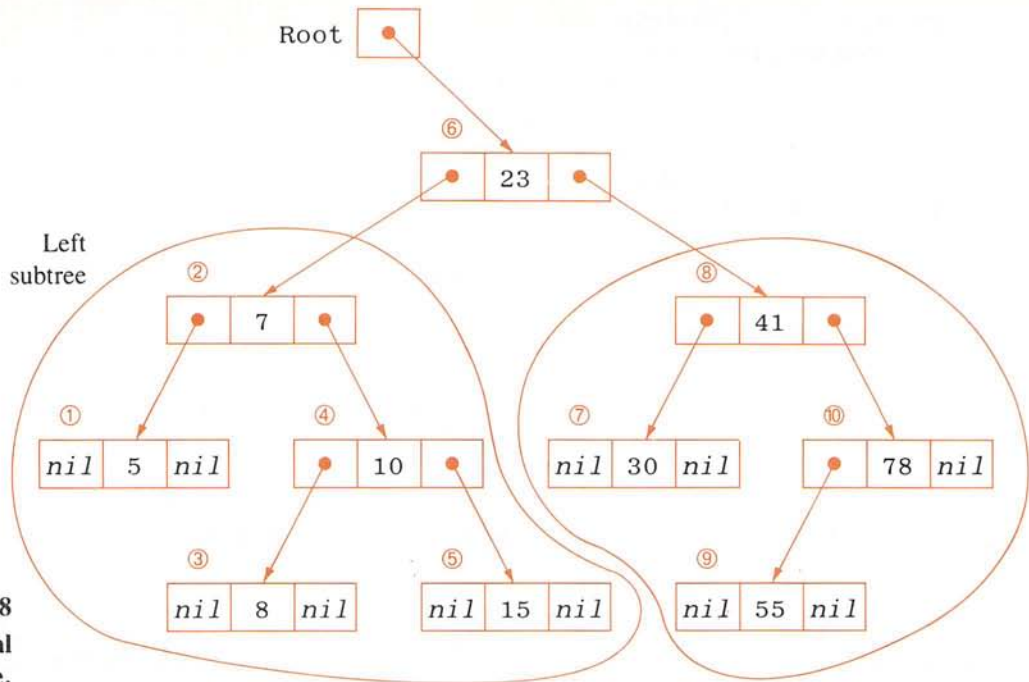


Figure 17.27  
A binary tree of  
the type used in  
the text examples.



**Figure 17.28**  
Inorder traversal  
of a tree.

*recursion  
on trees*

Figure 17.28 will help to explain why it is convenient to express tree algorithms recursively. Notice that each pointer emanating from the root node points to a smaller binary tree. In the figure these subtrees are circled and labeled *left subtree* and *right subtree*. Hence, algorithms can be stated very neatly in the form “do something to the root and apply the algorithm recursively once to the left subtree and once to the right subtree.” As an example, the following traversal algorithm is called *inorder traversal*:

**ALGORITHM—**  
*inorder traversal*

1. Traverse the left subtree.
2. Visit the root node (for example, to write out its contents.)
3. Traverse the right subtree.

*empty  
tree*

The above algorithm assumes that the tree is nonempty and does not specify what should be done with an empty tree. Yet recursively searching left and right subtrees will produce smaller and smaller subtrees until the algorithm is applied recursively to the empty tree, indicated by *nil*. At this point the algorithm should stop; to output the empty tree the instruction is “do nothing.” Hence, the complete procedure will execute the above algorithm only if the tree is nonempty. If the tree is empty, then after testing for and finding *nil*, the algorithm literally does nothing. Figure 17.29 shows the complete algorithm implemented as a procedure.

The circled numbers in Figure 17.28 indicate the order in which the nodes are processed by the inorder traversal procedure. Notice that the numbers are output in numeric order. We will come back to discuss this interesting observation shortly.

Two other common methods for traversing a tree are similar and correspond to permuting the three instructions in the recursive algorithm. If the root node is visited

```

type Pointer = ↑TreeNode;
   TreeNode = record
       Data: integer;
       Left, Right: Pointer
   end;

procedure OutputNodes (Root: Pointer);
{Outputs the integers in the nodes of the tree with root node Root↑;
if the tree contains numbers that satisfy the Binary Search Tree Storage Rule,
then they will be output in numeric order.}
begin{OutputNodes}
    if Root <> nil then
        begin{nonempty tree}
            OutputNodes (Root ↑ .Left); {Output left subtree.}
            write (Root ↑ .Data); {Output root node.}
            OutputNodes (Root ↑ .Right) {Output right subtree.}
        end {nonempty tree}
    end; {OutputNodes}

```

**Figure 17.29**  
**Procedure for**  
**inorder output**  
**of a tree.**

first, the algorithm is called *preorder traversal*. If the root is visited last, the algorithm is called *postorder traversal*. However, we will not discuss these alternative algorithms here except to note that they would traverse the nodes in a different order from the inorder traversal we coded in Figure 17.29.

Let us return to our inorder procedure for outputting the node contents and analyze why the nodes are output in numeric order. The reason is that we have stored the numbers in the tree in a special way known as the *Binary Search Tree Storage Rule*:

*binary*  
*search*  
*trees*

#### *Binary Search Tree Storage Rule*

1. All the numbers that are less than the number in the root node are in the left subtree.
2. All the numbers that are greater than the number in the root node are in the right subtree.
3. This rule applies recursively to the right and left subtrees.

As long as the numbers are stored in this way and we use the inorder traversal algorithm, then the root node number will always be output after all smaller numbers and before all larger numbers, and so the root node number is output in its correct location. Moreover, this applies to the root nodes of the subtrees and their subtrees and so on, until all nodes are accounted for and are seen to be output in their correct position. As the heading of the rule indicates, these sorts of trees are called *binary search trees*. The name is not derived from the fact that they can be output in this nice way but from another useful property they have, namely that they lend themselves readily to a binary search algorithm similar to the binary search algorithm we described for arrays in Chapter 14. The version for binary trees is given by the function in Figure 17.30.



```

type Pointer = ↑TreeNode;
      TreeNode = record
          Data: integer;
          Left, Right: Pointer
      end;

function TreeSearch(Query: integer;
                    Root: Pointer): boolean;
{Searches the binary search tree whose root node is pointed to by
Root. Returns true if the number Query is in some node of
the tree. Returns false if Query is not in the tree.}
begin{TreeSearch}
    if Root = nil then
        {empty tree}
        TreeSearch := false
    else if Query = Root↑.Data then
        {found Query}
        TreeSearch := true
    else if Query < Root↑.Data then
        {Search subtree to the left of the root.}
        TreeSearch := TreeSearch(Query, Root↑.Left)
    else {if Query > Root↑.Data then}
        {Search subtree to the right of the root.}
        TreeSearch := TreeSearch(Query, Root↑.Right)
    end; {TreeSearch}

```

**Figure 17.30**  
Function that  
searches a binary  
tree looking for  
a node.

#### ALGORITHM

The algorithm in outline form is

```

if Root = nil then
    the tree is empty and the number is not in the tree
else if the number is in the root node then
    the number is found
else if the number sought is less than the number in the root node then
    search the left subtree
else if the number sought is greater than the number in the root node then
    search the right subtree

```

This same basic method applies to retrieving data, such as names, that is stored in alphabetical order, or for that matter to any other type of ordered data.

The advantage of the binary search algorithm for trees is the same as it is for the binary search algorithm for an array: it is faster than other methods, such as searching a linked list or serially searching an array. One advantage of using trees rather than arrays is that trees may be of any size and may even change size during program execution, while an array is of a fixed size declared when the program is written.

## Case Study

### Building a Search Tree

#### Problem Definition

We wish to design a binary search tree that stores numbers according to the recursive rule we gave in the last section, so that we can later use the function `TreeSearch` (Figure 17.30) to search for numbers in the tree. For example, the numbers might be invalid credit card numbers that are read from a file or entered from the keyboard and then placed in a binary tree. For this problem we will assume they are read from a file of integers called `NumberFile`. The function `TreeSearch` can then be used to see if a given credit card is invalid. Since two cancellations of a credit card are the same as one cancellation, we will discard any repetitions and enter each number only once. Our goal can be summarized as follows:

*Postcondition: Root is pointing to the root node of a tree that:  
contains all the numbers in the file NumberFile,  
contains no repetitions of numbers, and  
satisfies the Binary Search Tree Storage Rule.*

#### Discussion

Searching a binary search tree is easy. Building one for a given collection of data is slightly more complicated but is still not too difficult. The idea of the algorithm is the same as that of the binary search algorithm implemented in Figure 17.30. Given a number to search for, that algorithm either finds the number or else it finds a *nil* pointer at the point where it expects to find a node that contains the number. Hence, if we already have a binary search tree and want to add a number, we can apply the same algorithm but end it slightly differently. If the number is found, the algorithm does nothing, since the number is already in the tree. If the number is not found, the algorithm replaces the *nil* pointer with a pointer that points to a new node containing the number. The algorithm follows this paragraph. To build a binary tree from a list of numbers, this algorithm can be used to add one node at a time, starting with the empty tree.

```

Input: A pointer Root and a Number to be inserted
if Root is nil then
    insert a new node containing Number
else if Number is in the node pointed to by Root then
    Number is already in the tree and so do nothing
else if Number is less than the number in the node pointed to by Root then
    insert Number in the left subtree by a recursive call
else if Number is greater than the number in the node pointed to by Root then
    insert Number in the right subtree by a recursive call
  
```

**ALGORITHM**  
for inserting a node

```

type Pointer = ↑TreeNode;
   TreeNode = record
       Data: integer;
       Left, Right: Pointer
   end;
   IntFile = file of integer;

procedure TreeInsert(Number: integer; var Root: Pointer);
{Precondition: Root points to the root node of a binary search tree.
Postcondition: If Number was not in the binary search tree, then a new node containing Number
has been added to the tree so as to preserve the Binary Search Tree Storage Rule.}
begin{TreeInsert}
    if Root = nil then
        begin{Insert Number in a new node pointed to by Root}
            new(Root);
            Root↑.Data := Number;
            Root↑.Left := nil;
            Root↑.Right := nil
        end {Insert Number in a new node pointed to by Root}
    else if Number < Root↑.Data then
        TreeInsert(Number, Root↑.Left)
    else if Number > Root↑.Data then
        TreeInsert(Number, Root↑.Right)
    {else if Number = Root↑.Data then
        do nothing since Number is already in the tree.}
    end; {TreeInsert}

procedure TreeBuild(var Root: Pointer; var DataFile: IntFile);
{Builds a binary search tree. Reads the integers for the nodes from the
file DataFile. DataFile has been opened with reset. Numbers in the file
may be in any order.}
var Next: integer;
begin{TreeBuild}
    Root := nil;
    while not eof(DataFile) do
        begin{Insert one integer}
            read(DataFile, Next);
            TreeInsert(Next, Root)
        end {Insert one integer}
    end; {TreeBuild}

```

**Figure 17.31**  
**Procedure to**  
**build a binary**  
**search tree.**



A Pascal procedure for building a binary search tree from a file of numbers is given in Figure 17.31. That procedure includes a call to the recursive procedure `Tree-Insert`, which implements the above algorithm.

The empty tree is represented by setting the `Root` equal to *nil*, so the first clause of the algorithm serves to start things off. However, it applies more often as the result of a recursive call when the search finds its way down to a *nil* marking an empty position in which to insert the node. Some people find the algorithm more intuitive if the clauses are read backwards. This is fine for intuition, but in fact is not correct, since if the pointer is *nil*, then it makes no sense to talk of left and right subtrees.

*empty  
tree*

The tree-building procedure in Figure 17.31 will always produce a binary search tree and, if the list of numbers is in random order, it will produce a balanced tree. A *balanced tree* is one in which all paths from the root node to the ends (the *nil*s) are of the same or almost the same length. In order to get all of the speed advantage of the binary search algorithm, the tree must be balanced or almost balanced. If the data arrives in an unfortunate sequence, then the tree will not be balanced, and the search will be slower. Techniques for making the tree balanced are quite complicated and are discussed in the references at the end of this chapter. In any event, the only issue is efficiency. The algorithm works for any sequence of integers.

*balanced  
trees*

---

“Would you tell me, please, which  
way I ought to go from here?”

“That depends a good deal on where  
you want to get to,” said the Cat.

*Lewis Carroll, Alice in Wonderland*

---

---

## Summary of Problem Solving and Programming Techniques

- Pointers provide a means for designing a wide variety of dynamic data structures, such as linked lists, stacks, and trees.
  - As one check to see that you have a pointer expression correct, check the type to see if it is a pointer type or a node type.
  - The advantages of a linked list over an array are that the linked list can grow and shrink and that it is easy to insert or delete a value (node) in the middle of a linked list.
  - A node (dynamic variable) can only be named by naming a pointer that points to the node. Unless a named pointer points to some key node(s) of a structure, the nodes in that structure can be lost to the program because the program has no way to refer to the nodes.
-

- The constant *nil* is used to mark endpoints, such as the end of a linked list, and to denote empty structures, such as an empty linked list or empty tree.
- Remember that *nil* is a pointer and not a node.
- Always check to see whether procedures to manipulate dynamic data structures, such as linked lists and trees, work correctly for the empty structure. Recursive procedures often terminate only when a series of recursive calls leads to an empty structure.
- Binary search trees can be used to store ordered data for rapid retrieval. Like a linked list, a binary search tree can grow and shrink in size. Like an array, a binary search algorithm can be used with the tree. Hence, in many situations, the tree combines the advantages of both arrays and linked lists.
- Research on trees in particular and dynamic data structures in general has produced a large number of nonobvious techniques for building and maintaining dynamic data structures. The references that follow include much more material on the subject.

---

## Summary of Pascal Constructs

### pointer types

Syntax:

$\uparrow$  <domain type>

Example:

```

type Arrow =  $\uparrow$ Node;
   Node = record
       Data: integer;
       Link: Arrow
   end;
var P1: Arrow;
```

The type for pointers that point to dynamic variables (nodes). In the example, the variable P1 is declared so that it can contain pointers to dynamic variables of type Node.

### naming the dynamic variable pointed to

Syntax:

<pointer variable>  $\uparrow$

Example:

P1  $\uparrow$

One way to name the dynamic variable pointed to by <pointer variable>.

### new

Syntax:

new (<pointer variable>)

---

Example:

```
new (P1)
```

Creates a new dynamic variable of the domain type of the <pointer variable> and leaves the value of the <pointer variable> pointing to this new dynamic variable.

### pointer variables in assignment statements

Syntax:

```
<pointer variable1> := <pointer variable2>
```

Example:

```
P1 := P2
```

Makes the value of <pointer variable1> point to the same thing as the value of <pointer variable2>.

### nil

Syntax:

```
nil
```

Predefined constant of a type that is compatible with pointers to any type of dynamic variable. *nil* does not point to any dynamic variable but is used to give a value to pointer variables that do not point to any dynamic variable (i.e., to give a value to "end points").

### dispose (Optional)

Syntax:

```
dispose (<pointer variable>)
```

Example:

```
dispose (P)
```

Releases the memory occupied by the dynamic variable pointed to by <pointer variable>. The value of <pointer variable> becomes undefined. The statement is undefined if <pointer variable> points to no dynamic variable. In particular, it is undefined if the value of <pointer variable> is *nil*.

### mark (Optional)

Syntax:

```
mark (<integer pointer variable>)
```

Example:

```
mark (M1)
```

Marks a place in the stack of dynamic variables so that all the memory allocated for dynamic variables after this point in time may later be released using *release*. *mark* and *release* are implemented in *TURBO Pascal* but not in standard Pascal.



**release** (Optional)

Syntax:

```
release (<integer pointer variable>)
```

Example:

```
release (M1)
```

Releases all of the memory that has been allocated to dynamic variables since the last call to `mark` using the same argument. `mark` and `release` are implemented in TURBO Pascal but not in standard Pascal.

---

## Exercises

### Self-Test Exercises

8. What is the difference between the kind of node used in a binary tree and the kind used in a doubly linked list?
9. Modify the type declaration in Figure 17.31 so that each node of the tree can have one more pointer that points to the node above it. The node above is often called the *parent node*, so call the new field `Parent`.

### Interactive Exercises

10. Using pencil and paper, simulate the procedure `OutputNodes` (Figure 17.29) on the tree in Figure 17.28.

### Programming Exercises

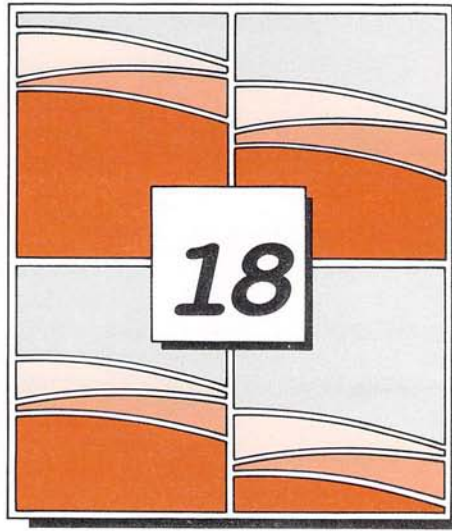
11. Write a procedure that takes as parameters a linked list of integers (the parameter will literally be a pointer to the head of the list) and two integers `L` and `U` such that `L` is less than `U`. The procedure should write to the screen all integers in the list that are between `L` and `U`. The list need not be sorted.
  12. Write a procedure that takes a (singly) linked list and reverses the order of the nodes in the linked list. For concreteness, make it a linked list of characters.
  13. A *queue* is what is often called a “line” in colloquial American English. It is a data structure that keeps a list of items, with new items added at one end and with items removed from the other end. For example, it might keep a list of names of people waiting for some limited resource, like a bank teller in a crowded bank. Queues are similar to stacks except that instead of a `Pop` operation, there is a `RemoveNext` operation, which takes nodes off the other end of the list. Like a stack, a queue can be implemented as a linked list. Write a program that uses a queue implemented as a linked list to keep track of people waiting in line for a bank teller or other resource. The user has a choice of either entering a new name to the queue or else asking for the next person to receive service. The program “waits in line” for the people so that they do not have to.
-

14. Redo Exercise 16 in Chapter 15, but this time use linked lists rather than arrays. With linked lists there is no limit to the number of digits in the two numbers being added.
15. Redo Exercise 19 in Chapter 15, but this time use linked lists rather than arrays. With linked lists there is no limit to the number of digits after the decimal point. Hence, this gives unlimited accuracy.
16. Write procedures to insert, find, and delete nodes in a doubly linked list. For simplicity, suppose that the nodes store integers.
17. A binary tree can be used to classify items according to a series of yes/no questions. Write a program that builds a tree to classify animals according to yes/no questions. The questions are stored in the nodes, and the answer determines which pointer to follow. Each leaf (end) node of the tree contains the name of an animal for which the yes/no answers are correct. For example, Is it very big? Does it eat meat? Does it have big ears? The answers *yes*, *no*, and *yes*, respectively, might lead to the name “elephant.” The program first constructs a tree with three levels of questions. It asks the user to input seven yes/no questions about animals. It then uses these seven questions as the questions in the tree. Each series will consist of three questions, and there will be eight possible sets of yes/no answers leading to animal names. The program displays each of the eight yes/no question sequences along with their answers, and then asks the user for an appropriate animal to enter. At that point the program offers to guess animals that the user is thinking of. It tells the user to think of an animal, and then asks the questions. The program works its way to an end node and then “guesses” the animal named in that node.
18. Redo the previous exercise, but this time have the program start with a very small tree with just one question—like Is it big?—and with two animals, like a mouse and an elephant. It then plays the game with the user, but whenever it guesses wrong (for example, if it guesses an elephant and the user says he/she was thinking of a dinosaur), it asks for a new question. For the two animals in this example, it would ask, “Please give me a question that will distinguish between an elephant and a dinosaur.” The user might give the question “Is it extinct?” The program then adds a new node with that question and adds the two animals below that node. In this way the tree gets larger as the game is played.
19. Write a recursive procedure that takes a linked list of integers, already sorted into ascending order, and produces a binary search tree that contains the same integers, stored according to the Binary Search Tree Storage Rule. The tree should be balanced. A tree is balanced if for each node, the two subtrees led to by its two pointer fields contain the same number of nodes (plus or minus one node).
20. Write a procedure that takes a binary search tree with integers stored according to the Binary Search Tree Storage Rule and copies them to a second tree that also satisfies this rule. The second tree should be as close to balanced as possible. The first tree may not have been balanced, and so this is a way to obtain a balanced search tree. A tree is balanced if for each node, the two subtrees led to by its two pointer fields contain the same number of nodes (plus or minus one node).

## References for Further Reading

- A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, 1983, Addison-Wesley, Reading, Mass. Covers more advanced topics.
- R.J. Baron and L.G. Shapiro, *Data Structures and Their Implementation*, 1980, Van Nostrand Reinhold, New York.
- P. Helman and R. Veroff, *Intermediate Problem Solving and Data Structures*, 1986, Benjamin/Cummings, Menlo Park, Ca.
- J.S. Rohl, *Recursion via Pascal*, 1984, Cambridge Computer Science Texts 19, Cambridge University Press, Cambridge and New York.
- D.F. Stubbs and N.W. Webre, *Data Structures with Abstract Data Types and Pascal*, 1985, Brooks/Cole, Monterey, Ca.
- A.M. Tanenbaum and M.J. Augenstein, *Data Structures Using Pascal*, Second Edition, 1986, Prentice-Hall, Englewood Cliffs, N.J.
-





## *Units for Modularization and Abstraction*

From mine own library with volumes that  
I prize above my dukedom.

*William Shakespeare, The Tempest*

## Chapter Contents

### Part I

Introduction to Units  
Writing a Simple Unit  
Predefined Units  
Interactive Exercises  
Version 4.0—Using Multiple Units  
Version 5.0—Using Multiple Units  
Interactive Exercises  
Units and Procedural Abstraction  
Owned Variables and the  
Initialization Section

Version 5.0—Uses Clause in  
Implementation Section (Optional)  
Unit Directories (Optional)

### Part II

Units and Data Abstraction  
Case Study—A Unit for a Queue  
Summary of Problem Solving and  
Programming Techniques  
Summary of TURBO Pascal Units  
Programming Exercises

**T**URBO Pascal has a mechanism called *units* which allows you to declare procedures and functions apart from any program. This allows you to build your own personal library of predefined procedures and functions. In this chapter we will introduce the notion of a unit and show how units can be used to facilitate procedural and data abstraction. This chapter applies only to TURBO Pascal. Units are not available in standard Pascal. A unit must be compiled to disk before it can be used. A unit is compiled to disk in the same way that a program is, and so you must know how to compile a program to disk before you can use units. If you do not already know how to compile a program to disk, then you should read Appendix 11 before reading this chapter. There are some slight differences in the details of handling units in versions 4.0 and 5.0 of TURBO Pascal. Readers using version 4.0 should read the sections labeled “Version 4.0” and skip the sections labeled “Version 5.0.” Readers using version 5.0 should do the opposite.

# PART I

---

## Introduction to Units

Pascal provides a number of predefined procedures and functions, such as `sqrt` and `round`. You need not include any declarations for these procedures and functions. They are defined for you. Suppose you wish to declare some additional procedures and functions of your own and add them to a library so that they would similarly be available to use in any programs that you write. So far we have no good way to build such a library. We could put each procedure into a file and read the file into our program when we need it, but that would clutter our program with unnecessary detail. There is a better solution. TURBO Pascal has a mechanism for building just such a library of your own personal predefined procedures and functions. The mechanism is called a *unit*. A *unit* is a collection of procedures, functions, defined constants, and defined types (and sometimes other items) which can be compiled apart from any program. The procedures, functions, constants, and types, which are defined in the unit, can then be used by any future program you write without having to be declared in the program.

Units have other advantages as well as allowing you to build up a library of procedures and functions. Since the procedures and functions in a unit are compiled separately ahead of time, the time to compile a program that uses such procedures and functions will be less than it would be if you had written the procedure and function declarations into the program. Units also allow you to write larger programs than you can write without using units. However, the time savings on any modest size program is not likely to be noticed, and even without units, you can write programs as large as most simple applications require. Hence, these two advantages of speed and size are not likely to concern most programmers. The main advantages of units are the one we have already discussed, namely building a library of procedures and functions, and one other advantage that will take a bit of explaining: As we will see, units provide a more effective way to practice procedural and data abstraction.

---

## Writing a Simple Unit

The procedure declarations which are defined by a unit are divided into two portions. One portion of the unit contains the headings for all the procedures and functions being defined for future use. This portion of the unit is called the *interface section*. It contains the procedure and function headings. It should also contain the header comments that explain what the procedures and functions do. That way the interface section contains all the information that a programmer who is using the procedures and functions needs to know. The remaining parts of the procedure and function declarations are placed in the other portion of the unit, which is called the *implementation section*. In other words, all local declarations, as well as the bodies of the procedures and functions being defined, are placed in the implementation section. The interface section de-

*interface  
section*

*implementation  
section*



scribes how the procedures and functions are used. The implementation section tells the compiler how to implement them.

Figure 18.1 contains a sample unit with the interface and implementation sections marked. This unit contains declarations for one commonly used procedure and one commonly used function. The procedure, which is called *Exchange*, is identical to the procedure of that name declared in Figure 5.1 and discussed at the beginning of Chapter 5. The procedure interchanges the values of two integer variables. The function, which is called *Power*, is identical to the function of that name which appears in Figure 8.3 of Chapter 8. As indicated in the header comment, the function computes

```

unit MyStuff;

interface
  procedure Exchange (var X, Y: integer);
    {Interchanges the values of X and Y.}

  function Power (X: real; N: integer): real;
    {Returns X to the power N. Precondition: N >= 0.}

implementation
  procedure Exchange (var X, Y: integer);
  var Temp: integer;
  begin{Exchange}
    Temp := X;
    X := Y;
    Y := Temp
  end; {Exchange}

  function Power (X: real; N: integer): real;
  var I: integer;
      Product: real;
  begin{Power}
    Product := 1;
    for I := 1 to N do
      begin{for}
        Product := Product*X
        {Product is X to the power I.}
      end; {for}
    Power := Product
  end; {Power}
end. {MyStuff}

```

Figure 18.1  
A unit.

powers. Notice that the procedure and function headings are repeated in the implementation section so that the implementation section contains complete procedure and function declarations.

The unit begins with the reserved word *unit* followed by the name of the unit and a semicolon. Any identifier, other than a reserved word, may be used for the unit name. In Figure 18.1 we used *MyStuff* as the name of the unit. Notice that the start of the interface section is indicated with the reserved word *interface*, without any semicolon. The start of the implementation section is indicated with the reserved word *implementation*, also without any semicolon. The entire unit ends with the reserved word *end* followed by a period.

*syntax*

It may seem redundant to include a procedure heading twice, and indeed it is. Although we do not advocate doing so, the TURBO Pascal compiler will allow you to omit a parameter list from the implementation section, whenever it is already included in the interface section. Hence, the following line in the implementation section of Figure 18.1

```
procedure Exchange(var X, Y: integer);
```

may be abbreviated to

```
procedure Exchange;
```

Similarly, the function heading in the implementation section of Figure 18.1 may be abbreviated to the following:

```
function Power;
```

However, these abbreviated headings are not the preferred style and we will not use them. Repeating the parameter list in the implementation section, as in Figure 18.1, produces an extra copy of the parameter list next to the procedure body where it can easily be referenced. This makes the implementation section easier to read.

A unit, such as the one in Figure 18.1, is compiled just as you would compile a complete program. While debugging the syntax of a unit, it makes sense to compile it to memory. However, before the unit can be used by a program or by another unit it must be compiled to disk. When it is compiled to disk, the unit's machine language code is in a file with the same first name part as the file containing the unit. The file with the machine language code receives the ending *.TPU*, while the file with the unit you write should end in *.PAS*. For example, if the unit is in the file *MyStuff.PAS*, then the code produced by the compiler will be placed in a file named *MyStuff.TPU*. *Normally, the file containing a unit must have the same name as the unit.* Thus, the unit *MyStuff* should be in the file *MyStuff.PAS*. When choosing unit names remember that the unit name will also be a file name and a file name can be at most eight characters long (plus a dot and extension such as *.PAS*).

*compiling  
a unit**file  
names*

After the unit has been compiled, the procedures and functions listed in the interface section of the unit can be used by any program. Any program that uses the unit must tell the compiler it is using the unit by inserting the reserved word *uses* and the unit name after the program heading. For example, if a program called *HotStuff* uses a procedure or function from the unit *MyStuff*, then the program would start

*uses  
clause*

```
program HotStuff;  
uses MyStuff;
```

This is called a *uses clause* and is separated from the rest of the program by a semicolon, as shown above.

---

## Predefined Units

*crt* TURBO Pascal comes with a number of predefined units. The one you are most likely to use is the unit `crt`. It contains a number of predefined procedures and functions to control screen output and keyboard input. The procedure `clrscr` is an example of one of the procedures in this unit. A call to `clrscr` will clear the screen. Other procedures and functions in this unit are described in Appendixes 13, 14, and 15. You do not need to write or compile this unit. It is already written and compiled for you. Nonetheless, you use it in the same way that you use any other unit. If you wish to use any of the procedures in this unit, then your program must contain a *uses* clause as in the following sample:

```
program Sample;  
uses crt;
```

The unit `crt` is available on IBM PCs, XTs, ATs, PS/2s, and true compatibles.

---

## Interactive Exercises

1. Change one of your programs by adding a call to `clrscr` as the first statement in the program body. This will clear the screen before the program dialogue begins. Do not forget to include a *uses* clause at the start of your program.
2. Write a unit with one procedure that writes "hello" to the screen. Compile the unit to disk. Write a program that tests the unit by calling your procedure so that the program writes "hello" to the screen.

---

## Version 4.0

---

### Using Multiple Units

A program can use more than one unit. If it does, then all of the units used are listed in the *uses* clause, separated by commas as in the following example:

```
program Sample;  
uses MyStuff, Unit2;
```

*uses  
clause*

All the units used by a program, even those used indirectly, should be listed in the *uses* clause. If `Unit2` uses a unit called `OldStuff`, then `OldStuff` should also

---



be listed in the *uses* clause of the program as in the following example. This is true even if the program itself does not include a call to anything defined in *OldStuff*.

```
program Sample;  
uses MyStuff, OldStuff, Unit2;
```

Units may be listed in any order, provided that each unit is listed before all the units that use it. For example, if *Unit2* uses *OldStuff*, then *OldStuff* must be listed before *Unit2*. Hence, the above ordering is legal, but the following is not:

```
program Sample;  
uses MyStuff, Unit2, OldStuff;  
{WRONG ORDER, if Unit2 uses OldStuff.}
```

If one unit uses another unit, then it must contain a *uses* clause. For example, let us continue to assume that *Unit2* uses the two units *OldStuff* and *MyStuff*, then *Unit2* must include a *uses* clause that tells the compiler that it uses them. This *uses* clause (if needed) is the first item in the interface section of a unit. Hence, *Unit2* would begin as follows:

```
unit Unit2;  
interface  
  uses OldStuff, MyStuff;  
  . . .
```

*units  
using  
units*

## Version 5.0

### Using Multiple Units

A program can use more than one unit. If it does, then all of the units used are listed in the *uses* statement, separated by commas as in the following example:

```
program Sample;  
uses MyStuff, Unit2;
```

Only the units actually used in the program need to be listed. If *Unit2* uses a unit called *OldStuff*, but the program itself does not include a call to anything defined in *OldStuff*, then there is no need to list *OldStuff* in the *uses* statement.

In version 5.0 of TURBO Pascal units may be listed in any order. Although it is not required, a good rule to follow is to list each unit before all the units that use it. For example, if *Unit2* uses *MyStuff*, then, unless you have explicit reasons for not doing so, it would be good practice to list *MyStuff* before *Unit2*. Since some earlier version of TURBO Pascal required this ordering, this ordering will make your programs more portable.

If one unit uses another unit, then it must contain a *uses* clause. For example, let us continue to assume that *Unit2* uses the two units *OldStuff* and *MyStuff*, then *Unit2* must include a *uses* clause that tells the compiler that it uses them. This

*units  
using  
units*

*uses* clause (if needed) is normally included in the interface section. It is the first item in the interface section. Hence, `Unit2` would normally begin as follows:

```
unit Unit2;  
interface  
  uses OldStuff, MyStuff;  
  . . .
```

---

## Interactive Exercises

3. Write a program that clears the screen and then writes “hello” to the screen. The program should use the unit `crt` and the unit you wrote for Interactive Exercise 2.
4. Write a procedure that clears the screen and then writes “Hello from a clean program” to the screen. This procedure will include a call to the procedure `clrscr`. Write a program to test the procedure. The program will include a *uses* clause for the unit `crt`. Next place your procedure in a unit, compile the unit to disk, and write a test program to test this unit by calling the procedure in the unit. You will then have a program that uses a unit and this unit will, in turn, use the unit `crt`.

---

## Units and Procedural Abstraction

*public  
declarations*

The interface section of a unit is not limited to containing only procedure and function headings. It may contain type, constant, or any other kind of declarations. For example, the interface section of the unit in Figure 18.2 contains a declaration for the defined constant `Rate`. Everything declared in the interface section is *public* information and may be referenced by any program (or unit) that uses the unit. Information that you wish to hide from the main program must be placed in the implementation section.

*private  
declarations*

In addition to those procedures and functions whose headings appear in the interface section of a unit, the unit may also have other procedure, function, constant, and/or other declarations which are only in the implementation section and are not mentioned at all in the interface section. These declarations are sometimes called *private* declarations. They are local to the unit and cannot be referenced by any program (or other unit) which uses the unit. For example, if a program uses the unit `MoneyOut` in Figure 18.2, then it can use the procedure `WriteInMoney`, the procedure `WriteMoney`, and the constant `Rate`. However, it cannot reference either the constant `Blank` or the function `Width` which appear in the implementation section. `Blank` and `Width` are local to the unit `MoneyOut`. They can be used in the unit `MoneyOut`, but they have no meaning outside of that unit.

*public and  
private  
information*

The interface section of a unit is the *public* section and the implementation section is the *private* section. A program that uses the unit may reference anything in the interface (public) section of the unit, but it may not reference anything in the implementation (private) section. This ability to have public and private sections is one good method for enforcing procedural abstraction. Recall that procedural abstraction is a kind of selective forgetting. After a procedure is written, the user of a procedure should

```

unit MoneyOut;
{Declares procedures to output money amounts in dollars and cents notation.}

interface
    const Rate = 0.10;

    procedure WritelnMoney(Amount: real);
    {Outputs Amount in dollars and cents notation and then
    advances to the next line in the same manner as writeln does.}

    procedure WriteMoney(Amount: real);
    {Outputs Amount in dollar and cents notation. Prints a
    trailing blank. Does not advance to the next line.}

implementation
    const Blank = ' ';
    function Width(Amount: real): integer;
    {Returns the total field width needed to output Amount in dollars and cents notation.}
    var Dollars, Count: integer;
    begin{Width}
        Dollars := trunc(Amount);
        Count := 0;
        while Dollars > 0 do
            begin{Drop one digit}
                Dollars := trunc(Dollars/10);
                {For example, this changes 198 to 19.}
                Count := Count + 1
            end; {Drop one digit}
        Width := Count + 3
        {+1 for the decimal point + 2 for the cents = +3}
    end; {Width}

    procedure WritelnMoney(Amount: real);
    begin{WritelnMoney}
        writeln(Blank, '$', Amount: Width(Amount):2)
    end; {WritelnMoney}
    procedure WriteMoney(Amount: real);
    begin{WriteMoney}
        write(Blank, '$', Amount: Width(Amount):2)
    end; {WriteMoney}
end. {MoneyOut}

```

**Figure 18.2**  
A unit with private  
declarations.



only be concerned with *what* the procedure does; he or she should not need to remember anything about *how* it is done. In order to use a procedure, all that the programmer should need to know about are the procedure heading and the header comment that explains what the procedure does. The programmer using the procedure should not need to know what local procedures are used, or what local variables are used, or any other details about how the procedure is implemented. With units this process of selective forgetting is enforced by *hiding* those details in the implementation section. This keeps the programmer's thinking uncluttered and provides for a well defined communication interface between procedures.

If you did not have units available, it would be much more difficult to “hide” declarations such as `Blank` and `Width` in Figure 18.2. If you did not have units, you could make them local to each of the two procedures `WriteInMoney` and `WriteMoney`. However, that would be a much less desirable way of hiding information. Local procedures and functions make a program difficult to read since it is difficult to keep track of what procedures are local to what other procedures. By using units, you avoid local procedures within procedures but still obtain all the procedural abstraction advantages they would afford. Moreover, by using units you can have a group of declarations that you need write only once, and that can be local to a whole group of procedures. In the unit `MoneyOut`, there is only one declaration for each of `Blank` and `Width` and yet they are used by both of the procedures `WriteInMoney` and `WriteMoney`. Since they are entirely within the implementation section, they behave as if they were local to both procedures. Units allow you to hide information in a very clean manner.

---

## Owned Variables and the Initialization Section

An implementation section may contain any type of declaration, including a variable declaration. A variable that is declared in the implementation section of a unit is very much like a local variable that can be shared by all the procedures and functions in the unit (both the public and the private ones). We will use the term *owned variables* to refer to such variables. (In the programming literature you will find owned variables referred to in a wide variety of ways including *own variables*, *private variables*, *static variables*, and *closed variables*.) As we will see, owned variables can be used for much more than simple local variables can be used for. We will illustrate this use with a simple example.

Suppose that you are head of software development for the famous Häägiñ-RøßBén chain of ice cream stores. Despite their old world image they have decided to computerize all their stores. You are in charge of the software engineering team that is designing a general-purpose computer program to serve multiple purposes including calculating change, keeping track of the company's  $10^{39}$  flavors, and even replacing the gizmo that gives out tickets showing customers' turns. For this example we will focus on replacing the turn-giving gizmo. The gizmo gives out slips of paper with numbers printed on them. It starts with the number 1, then 2, 3, and so forth up to 100 and then starts over again. So the numbers eventually are 99, 100, 1, 2, 3, etc. Customers are served in

---

the order in which they receive numbers. What you would like for your program is a function called `GetNumber` that will return such numbers. The first time that `GetNumber` is called in the program it should return 1, the next time it should return 2, and so forth.

There is one big problem with designing the function `GetNumber`. Every time the function is called it must return the number that is one more than the number it returned the previous time. Hence, that previous number must be remembered somehow. Your first attempt at designing the function is to use a global variable called `Next` which will contain the next number to be given out. The program initializes `Next` to 1 and then the function increments the value of this global variable. Your first attempt at a function declaration is the following:

```
type Turn = 1..100;
function GetNumber: Turn;
begin{GetNumber}
  GetNumber := Next;
  Next := (Next mod 100) + 1
end; {GetNumber}
```

This is a function with side effects, which generally speaking is a bad idea. However, in this case some sort of side effect is inevitable. That number must be remembered and whatever remembers it will be a side effect of some sort. But there is an even more serious problem.

You want to assign the rest of the programming of this large program to your assistant programmers, but you have a problem with them. These programmers are all ice cream fans, but they hate to wait in line. You want to be sure that they cannot change the global variable by some trick so that they can go to the front of the line. For example,

```
if Customer = 'Savitch' then
  Next := 1
```

What you need to do is to somehow make this global variable inaccessible to the main program. That is exactly what *owned* variables were designed for.

To make the variable `Next` inaccessible to the main program, you can declare it in the implementation section of a unit called `Hidden`. The unit is shown in Figure 18.3. The programmers can use the unit, but they cannot refer to anything in the implementation part of the unit and so they cannot reference the variable `Next`. A variable such as `Next` is called an *owned* variable because it is the private property of the unit.

An owned variable such as `Next` combines the desirable properties of both local and global variables. Like a local variable, the program (outside of the unit `Hidden`) cannot change the variable. It is like a variable that is local to the unit. However, a true local variable would lose its value after each call of the function is completed. An owned variable such as `Next` retains its value so that the next time the function is called the variable will have the value that was left by the last function call. In this way it is like a global variable that is forbidden to everything but the unit in which it is declared.

*owned  
variables*

*saved  
values*

```

unit Hidden;
interface
  type Turn = 1..100;
  function GetNumber: Turn;
  {Function of zero arguments that returns the numbers 1 through 100 in order.
   The first number returned in a program is 1. Each call returns the next number.
   Wraps around at the end so that the number after 100 is 1.}

implementation
  var Next: Turn; {Owned variable to remember the next number
                  to be returned by GetNumber.}

  function GetNumber: Turn;
  begin{GetNumber}
    GetNumber := Next;
    Next := (Next mod 100) + 1
  end; {GetNumber}

begin {initialization}
  Next := 1
end. {Hidden}

```

**Figure 18.3**  
A unit with an  
owned variable.

#### initialization section

The unit `Hidden` has one other new feature that we needed for this project. Since the variable `Next` cannot be referenced in the main program, it cannot be initialized in the main program. To cope with these sorts of problems, units can optionally contain an initialization section. In the unit `Hidden` in Figure 18.3, the initialization section is the section between the line

```
begin{initialization}
```

and the final `end`.

If you insert the identifier *begin* just before the final *end* in a unit, then you have added an *initialization section*. Between that *begin* and *end* pair, you can insert any Pascal statements. These statements will all be executed before the execution of the main body of any program which uses the unit. In this case we inserted an assignment statement to initialize the variable `Next` to 1.

The initialization section of a unit is very much like the main body of a program. It contains a list of Pascal statements to be executed. These statements are executed at the start of any program (or unit) which uses the unit. As the name suggests, the initialization section is designed to be used to set initialization conditions. It is typically used to initialize variables, as we did with `Next` in our example, to open files, and to do any other tasks which are needed in order to set the stage for the use of the procedures and functions defined in the unit.



---

## Version 5.0

---

### Uses Clause in Implementation Section (Optional)

In version 5.0 a *uses* clause may be placed in the implementation section. This allows for additional hiding of information. For example, suppose that UnitA uses UnitB, then UnitA must contain a *uses* clause that lists UnitB. This *uses* clause may be placed in either the interface section or the implementation section. If the *uses* clause is placed in the implementation section then the declarations (for example, type declarations) given in UnitB can only be used in the implementation section of UnitA. They cannot be used in the interface section of UnitA. This is seldom an important issue. In the next paragraph we discuss a more important reason for placing a *uses* clause in the implementation section. When it is placed in the implementation section, a *uses* clause is placed immediately after the reserved word *implementation*. For example, the layout of UnitA could be as follows:

```
unit UnitA;  
interface  
  . . .  
implementation  
  uses UnitB;
```

Version 5.0 permits circular use of units. It is possible for UnitA to use UnitB and for UnitB to use UnitA. In such cases, the *uses* clauses must be placed in the implementation sections. This is the most common reason for placing a *uses* clause in the implementation section of a unit. If you are writing units with circular reference, you should use the **Make** command on the **Compile** menu rather than the **Compile** command. With the **Make** command **TURBO** will manage your units for you, automatically compiling any units that need to be compiled or recompiled. Hence, if you use **Make** to compile one of two units that reference each other, that single **Make** will compile both of the units.

---

### Unit Directories (Optional)

Most of the common predefined units, such as `crt`, are in the file `TURBO.TPL`. This file is loaded into the computer's memory when you load the **TURBO** Pascal system and so these units are always available. No matter what the active disk drive is or what the active directory is, your program can use any of these units and the system will find the unit. However, if you write a unit of your own and later use the unit in a program, the **TURBO** system ordinarily expects to find that unit in the active directory of the active disk drive. If you have a unit with procedures that you use often, this means that you must move the unit every time you change active directories, or else you will lose ac-

*TURBO.TPL*

cess to that unit for as long as it is not in the active directory. If you have developed a sizeable library of units, this can be a nuisance. TURBO provides two ways to make the units that you write as readily available as the predefined units in TURBO. TPL. One method is to add your units to the file TURBO. TPL. This can be done with the TURBO utility program TPUMOVER. Instructions for using TPUMOVER can be found in the TURBO manual and we will not describe them here. However, we will discuss one other method.

*Options*  
*Directories*  
*Unit directories*

The other method for making your units always available is to tell the TURBO system what directory the units are in. For example, let us suppose that you have all your totally debugged units in the directory MyUnits on disk drive B and you want to tell the system to always look in there to find units. First activate the Options pull-down menu, then choose the Directories command on that menu. That will produce a list giving the types of directories you can specify. Choose Unit directories from that list and yet another small window will appear on the screen. This window is the system's way of asking for the name of a directory. Simply type in the disk drive and full path name of the directory that contains all your units. Whatever you type in will appear in the window, as in the following sample:

Unit Directories
B: \MyUnits

After typing in the directory, press return. The small window will disappear and the directory name will be on the Directories menu immediately after the entry Unit directories. The system has now been told to look in this directory to find the units which you have written. When it encounters a *uses* clause, the system first looks in TURBO. TPL to find a unit, since the most common predefined units, such as crt, are normally in there. If it does not find the unit in TURBO. TPL, then it looks in the active directory of the active disk drive. If the system does not find the unit in either of those locations, then it looks in the directory which you named as the Unit directories. In our example, that means that it would look in the directory MyUnits on disk drive B. If that fails, it gives up and issues an error message.

Notice that, even though you have named a directory to hold fully debugged units, you can still use the active directory to store and develop new units or even new versions of existing units. If the system can find a unit in the active directory, it will not look in the Unit directories which you specified.

If you wish to change the directory you use to hold units, then simply repeat the procedure we described above, but type in the name of a different directory.

*multiple*  
*directories*

This all works fine provided that all your units are in one directory, but what if you have them organized into several different directories? In that case you can list all the directories, separated by semicolons, as in the following example:

Unit Directories
B: \MyUnits; B: \OurUnits; C: \Lib \Class

In this case, TURBO will, as always, first look in TURBO.TPL and then in the active directory to find a unit. If that fails it looks in the directory MyUnits on disk drive B. If the unit is not there, it will then look in the directory OurUnits on disk drive B. If it still cannot find the unit, it looks in the directory Lib\Class on disk drive C. If it fails there, it gives up and outputs an error message.

Until you become intimate with the TURBO system it is safest to keep the .PAS source file for a unit and the .TPU file with the machine code version of the unit together in the same directory. That way, whichever version the system is looking for (or you are looking for) will be where you think it is. When you become familiar with how the system works you may want to arrange things differently. For example, if you are compiling programs by choosing the Compile command from the Compile menu, then you only need to keep the .TPU files in your Unit directories. If you need space and if you compile your programs in this way, then you can move the .PAS files to some other location for storage.

## PART II

---

Ignorance is preferable to error; and  
he is less remote from truth who  
believes nothing, than he who believes  
what is wrong.

*Thomas Jefferson, Notes on the State of Virginia*

---

---

## Units and Data Abstraction

Procedural abstraction and data abstraction go hand-in-hand. Each is a judicious kind of forgetting. If we are using procedural abstraction, then, after a procedure is written, we can and should concentrate on what the procedure does and can safely “forget” how it does it. In the case of data abstraction we apply a similar technique to data structures. Data abstraction consists of concentrating on the essential details of what a data structure is and ignoring (i.e., “forgetting”) the details about exactly how it is implemented. Earlier in this chapter we saw how units can be used to enforce procedural abstraction by placing the details about how a procedure works into the implementation section of the unit. By placing such detail in the implementation section we make that detail inaccessible to any program which uses the unit and so enforce the selective “forgetting” which underlies procedural abstraction. In the next case study we will see how units can be used to enforce data abstraction in a similar manner. In order to set the stage for this discussion, it would be a good idea to read (or reread)

---





hours are particularly crowded. When a student arrives, the student places his or her student number in the queue. When the instructor has finished talking with one student and is ready to talk to another student, the instructor takes the next available number from the queue and announces that that student is next.

## Discussion

The principle of data abstraction as applied to units says that those properties of our queue which are essential to its use should be placed in the interface section of the unit and the details of exactly how these properties are realized should be hidden in the implementation section. In order to get a feel for the sort of detail that should be hidden we will do this example backwards and start with the implementation section. However, it is usually preferable to start with the interface section and, after this first example, you should adopt that order of doing things.

In this example, we will implement the queue as an array of integers declared as follows:

```
const MaxQSize = 100;
type Index = 1..MaxQSize;
    ExtendedIndex = 0..MaxQSize;
    QArray = array[Index] of integer;
var Q: QArray;
    Last: ExtendedIndex;
```

The integers in the queue will be stored in  $Q[1]$ ,  $Q[2]$ ,  $Q[3]$ , . . . through  $Q[Last]$ . Whenever we add a number to the queue, we will place it in location  $Q[Last+1]$  and will increase the value of  $Last$  by one so that the value of  $Last$  is always the index of the last number in the queue. When we remove a number from the queue, we always remove the number in  $Q[1]$  and then move all the numbers down one so that the value of  $Q[2]$  moves to  $Q[1]$ , the value of  $Q[3]$  moves to  $Q[2]$ , and so forth. However, all this detail will be placed in the implementation section. The user of the queue need not know the index type of the array. For that matter, the user of the queue need not even know that the queue is implemented as an array.

The details that the user of the queue needs will go in the interface section of the unit. What does a programmer need to know about in order to write a program that uses the queue? The programmer needs some way to add numbers to the queue and some way to take the next available number out of the queue. We will provide a procedure called `Add` which performs the first task and a function of no arguments called `Next` which returns the next available number.

For example, in order to add the numbers 8, 9, 2, and 7 to the queue the following procedure calls should suffice:

```
Add(8); Add(9); Add(2); Add(7)
```

In order to allow the program to test to see if a number can be added to the queue, we will provide a boolean valued function `FullQ` which returns `true` if the queue is full and `false` if there is room to add a number to the queue. `FullQ` has zero arguments.

*implementation  
section*

*interface  
section*

`Next` will be a function of zero arguments which returns the next number in the queue (i.e., “the first number in line”). For example, the following will write out the first number in the queue:

```
writeln(Next)
```

A call to `Next` will return the first number in the queue and will also have the side effect of removing that number from the queue. For example, if the above three calls to `Add` represent the only numbers added to the queue, then the above `writeln` will output 8, and the number 9 will then become the first number in the queue. The next call to `Next` will return 9.

In order to allow the program to test to see if there are any numbers at all in the queue, we will provide a boolean valued function `EmptyQ` which returns `true` if the queue is empty and `false` if there is at least one number in the queue. `EmptyQ` has zero arguments.

The description of the procedure `Add` as well as the descriptions of the functions `Next`, `FullQ`, and `EmptyQ` will be placed in the interface section of our unit, but all details about how they work will be hidden in the implementation section. Notice that our discussion of `Add` and `Next` made no mention of arrays. All details about arrays will be hidden from the user. The user of a queue is adding and deleting integers in an abstract data structure which may be implemented as an array or may be implemented in some other way. The user should neither know nor care about this detail.

The unit that implements our queue of integers is shown in Figure 18.5. Notice that the array `Q` is an owned variable declared in the implementation section and so it is not available to any program that uses the unit. If a program uses this unit, it can use

owned  
variables

```
unit Queue;
{Provides a queue of integers and procedures to manipulate the queue.}

interface
  function FullQ: boolean;
  {Returns true if the queue is full.}

  function EmptyQ: boolean;
  {Returns true if the queue is empty.}

  procedure Add(Element: integer);
  {Adds Element to the end of the queue.
  Precondition: the queue is not full.}

  function Next: integer;
  {Returns the next available element from the queue. Removes
  that element from the queue. Precondition: EmptyQ is false.}
```

Figure 18.5

A queue as a unit.



```

implementation
  const MaxQSize = 100;
  type Index = 1..MaxQSize;
        ExtendedIndex = 0..MaxQSize;
        QArray = array[Index] of integer;
  var Q: QArray;
        Last: ExtendedIndex;

  function FullQ: boolean;
  begin{FullQ}
    FullQ := (Last = MaxQSize)
  end;{FullQ}

  function EmptyQ: boolean;
  begin{EmptyQ}
    EmptyQ := (Last = 0)
  end;{EmptyQ}

  procedure Add(Element: integer);
  begin{Add}
    if FullQ then
      writeln('Error: Queue overflow.')
    else
      begin{Add Element}
        Last := Last + 1;
        Q[Last] := Element
      end{Add element}
    end; {Add}

  function Next: integer;
  var I: Index;
  begin{Next}
    if EmptyQ then
      writeln('Error: Accessing an Empty Queue.')
    else
      begin{Return next and delete from queue.}
        Next := Q[1];
        for I := 1 to (Last - 1) do
          Q[I] := Q[I + 1];
        Last := Last - 1
      end{Return next and delete from queue.}
    end;{Next}

  begin{initialization}
    Last := 0
  end.{Queue}

```

**Figure 18.5**  
A queue as a unit.

the procedure `Add` and the functions `EmptyQ`, `FullQ`, and `Next`, but it would be illegal for the program to have any direct reference to `Q` such as

`Q[1] := 57 {ILLEGAL in any program using the unit Queue.}`

Notice that all the constant, type, and variable declarations are in the implementation section and so are not available to the program. Also notice that the variable `Last` is initialized in the initialization section. Since it is not available to the program, it must be initialized in the unit.

As we indicated earlier in this chapter, owned variables combine the best features of local and global variables. The array `Q` and the variable `Last` offer us another good example of the use of owned variables. Like local variables, they are not accessible to any program that uses the unit. Like global variables, they retain their values from one procedure call to another.

Before we leave this case study, let us take one more look at the unit `Queue` to see if it truly embodies the principles of data abstraction. If it does, then we should be able to change the implementation of the queue and it should have no effect on any program that uses the unit. If you study the unit, you will see that we can safely change the implementation. We could rewrite the implementation section to use some other index type and none of the programs which use the unit would need to be changed. We could even rewrite the unit so that the queue is implemented without using an array at all and the programs would still work without any changes. In Chapter 17 we discussed a data structure called a “linked list.” If you read that chapter, it might occur to you that queues are easily implemented using these linked lists (whatever they are). If that happens, then you can rewrite the unit `Queue` using a linked list rather than an array. If you do so, all of the programs which you wrote in the interim and which use this unit will still work. Rewriting the unit will not require any changes to the programs which use the unit. That is what data abstraction is all about.

---

We all know—the *Times* knows—but we pretend  
we don't.

*Virginia Woolf, Monday or Tuesday*

---



---

## Summary of Problem Solving and Programming Techniques

- Units may be used to build your own library of additional predefined procedures and functions.
  - Units can be used to enforce procedural abstraction. This is done by placing the details about how a procedure is implemented into the implementation section of a unit and thereby “hiding” these details.
  - Units can be used to enforce data abstraction. This is done by placing the details
-

*unit* <Name of unit>;

*interface*

*uses* <List of units used by this unit>; {*Omit if no units are used.*}

<Headings for procedures and functions

to be used outside of the unit;

any other public declarations>

*implementation*

<The complete procedure and function declarations for

those with only headings in the interface section;

all private declarations including any owned variables.>

*begin*{*not needed if there are no initialization statements*}

<Initialization statements consisting of Pascal statements  
that may reference anything in either the implementation or  
interface sections.>

*end.*

**Figure 18.6**  
**Syntax for a unit.**

about how a data structure is implemented into the implementation section of a unit and thereby “hiding” these details.

- Owned variables can combine the best features of global and local variables. Like global variables, they retain their values after a procedure call is completed. Like a local variable, they are inaccessible outside of the unit in which they are declared.

---

## Summary of TURBO Pascal Units

Figure 18.6 contains a template for a unit. The unit would normally be in a file named

<Name of unit>.PAS

Before any program or other unit can use this unit it must be compiled to disk. Compiling the unit to disk will place the translated machine language code in the file

<Name of unit>.TPU

---

## Exercises

### Programming Exercises

5. Design a unit called `Just2` which defines three functions that can be used by your programs. The three functions are described by the following headings which should appear in the interface section of the unit:



```

function Max(N1, N2: integer): integer;
{Returns the larger of N1 and N2.}

function Min(N1, N2: integer): integer;
{Returns the smaller of N1 and N2.}

function Ave(N1, N2: integer): real;
{Returns the average of N1 and N2.}

```

Write a program to test the unit.

6. Do Exercise 18 in Chapter 8. Place the function in a unit so that programs can use the function without having to include the function declaration. Write a test program to test the unit.

7. Design two functions, one that returns the average of four scores and the other that returns the standard deviation of four scores. The standard deviation is defined to be the square root of the average of the four values  $(s_i - a)^2$  where  $a$  is the average of the four scores  $s_1, s_2, s_3$ , and  $s_4$ . The function to compute the standard deviation should call the function that computes averages. Place the declarations in a unit so that a program can use either or both functions. Write a program to test the unit.

8. Do Exercise 20 in Chapter 8. Place the function in a unit so that programs can use the function without having to include the function declaration. Write a test program to test the unit.

9. Design a unit called `BigChar` which has two parameterless procedures called `WriteYes` and `WriteNo`. `WriteYes` writes the word "YES" to the screen in large letters made up of asterisks. `WriteNo` writes the word "NO" to the screen in large letters made up of asterisks. The letters should be at least five times the size of normal letters. Write a test program to test the unit.

10. Write a procedure that has one value parameter of type `char`. The procedure writes "YES" to the screen if the parameter value is either 'y' or 'Y' and writes "NO" to the screen if the parameter has any other value. Place the procedure in a unit called `Echo`. The unit `Echo` will use the unit `BigChar` designed in the previous exercise. Write a test program to test the unit `Echo`.

11. Design a unit called `Degrees` which includes versions of the three trigonometric functions given in Figure 8.8 (the first three entries in the table) but which work in degrees rather than radians. Name the functions `Darctan`, `Dcos`, and `Dsin`. Write a program to test the unit.

12. (This exercise uses the optional sections on random number generators that are in Chapter 8.) Design a unit called `RNumbers` which has declarations for new versions of the random number generators `Random` in Figure 8.9 and `RandomReal` at the bottom of page 287. In these versions the global variable `Memory` is replaced with an owned variable so that it is not accessible to any calling program. Hence, these new random number generators will be functions with no arguments. Include a new procedure called `Reset` with the following heading and meaning:

```

procedure Reset(Seed: integer);
{Resets the random number generators with the seed value Seed.}

```

This procedure resets the variable `Memory` to the value of `Seed`. Thus if a program wants to start its random number generator with a seed value equal to `Seed`, then there should be a call to `Reset (Seed)`. However, this is the only way to access the variable `Memory`. Notice that, even though a program can change the value of `Memory`, there is still value in making it an owned variable. Since it is an owned variable, it need not be an argument to the functions `Random` and `RandomReal`. Moreover, there is no danger that the programmer will inadvertently reset the variable `Memory`.

13. Write a unit called `Nim` for playing the game of Nim which is described in Exercise 27 of Chapter 8. The size of the three piles should be kept as the values of three owned variables. The unit will include a procedure called `Move` with the following heading and meaning:

```
procedure Move(Pile, Number: integer; var Legal: boolean);  
{Executes a move consisting of picking up Number sticks from pile number Pile,  
provided that is legal. If the move is legal, then Legal is set to true, otherwise it  
is set to false. The result of the move is displayed on the screen.}
```

There should also be a procedure called `StartNim` that has no arguments and that initializes the game. Both the procedures `Move` and `StartNim` should produce a nice display of the game similar to the sample shown at the top of page 328. In the easy version of this exercise, the piles always start out with the values 3 sticks, 5 sticks, and 7 sticks. In the difficult version, the piles are initialized by a call to a random number generator that sets each pile of sticks equal to a randomly chosen value that is at least 2 and at most 10 sticks. If you use a random number generator and you have already done the previous exercise, then use the unit from that exercise to obtain your random number generator.

14. The implementation of `Next` in the unit `Queue` in Figure 18.5 is not very efficient. Whenever a number is deleted from the queue, all the remaining numbers must be moved. Rewrite the implementation section of the unit so that no number in the array `Q` needs to move when a number is deleted. To do this you can use an additional owned variable called `First` which is the index of the “first number in line.” The variable `Last` will be the index of the “last number in line.” A number is added as in Figure 18.5. To delete an integer, all that is needed is to increment the variable `First`. To make this all work out, your unit will have to treat the array as if the numbers were arranged in a circle. So that when you increment `MaxQSize`, the result is 1, rather than `MaxQSize + 1`. You may want to add a private function to perform this incrementing.

15. (This exercise uses the material on linked list in Chapter 17.) Rewrite the implementation section of the unit `Queue` in Figure 18.5 so that the queue is implemented as a linked list using pointers, rather than as an array.

16. (This exercise uses the material on linked lists and stacks given in Chapter 17.) Write a unit called `Stack` which implements a stack using a linked list. It should have procedures for `Push` and `Pop` as described in Figure 17.18. These procedures will be

similar to those given in Figure 17.19, but they will not have the parameter `Top`. `Top` will instead be an owned variable. There should also be a boolean valued function `EmptySt` which returns `true` if the stack is empty and returns `false` otherwise. `EmptySt` is similar to the function `Empty` in Figure 17.19, but `EmptySt` has no arguments.

---



# Appendix 1

## The goto and exit Statements

The term *flow of control* refers to the order in which the statements and substatements of a program are executed. There is one Pascal mechanism for flow of control that we have not yet discussed. That mechanism is the *goto statement*. The method of using this statement, in fact the very question of whether or not it should be used at all, is very controversial. In this appendix we briefly explain the *goto* statement and the controversy surrounding it.

As an example, consider the following program fragment:

```
writeln('First Statement');  
42: writeln('Statement Labeled 42');  
    writeln('Third Statement');  
    goto 42;
```

The number 42 on the second line is called a *label*. It has no effect on the statement. It is just a way of giving a name to the second `writeln` statement.

The *goto* instructs the computer to next execute the statement labeled 42. After executing the statement labeled 42, the computer proceeds to the next statement after that; in other words, the computer forgets whether it arrived at a labeled statement via a *goto* statement or by some other means.

Thus, the above example is an infinite loop with the output

```
First Statement  
Statement Labeled 42  
Third Statement  
Statement Labeled 42  
Third Statement  
Statement Labeled 42  
Third Statement
```

The last two lines are repeated indefinitely.

In standard Pascal all labels, such as 42, must be integers in the range 0 to 9999. However, they are used only as names of statements and not as numbers. In particular, a

---

*goto* statement cannot contain an integer variable. In TURBO Pascal you also may use any nonreserved word identifier as a label. All *labels* must be declared. The label declarations come before all other declarations in a block. The syntax consists of the reserved word *label* followed by a list of labels; the labels are separated by commas and terminated with a semicolon. For example, the following declares 100 and 42 to be labels:

```
label 100, 42;
```

Once a label is declared, it may be used to label a statement and then used in a *goto* statement. A *goto* statement consists of the identifier *goto* followed by a label. The identifier *goto* is a single word with no spaces. The execution of a *goto* statement is frequently called a *jump* because the execution “jumps” to the labeled statement specified after the *goto*.

There is one important restriction that applies to the use of *goto* statements. It is possible to use a *goto* statement to jump out of a structure such as a loop or procedure. However, a *goto* statement may not be used to jump into a structure. If a *goto* is used to jump into a structure, the effect is undefined and unpredictable.

The *goto* has a long history; at least, it is long when compared to other things in the young field of computer science. Although it was usually spelled differently, the *goto* was an important feature in virtually all early programming languages. Machine languages invariably have *goto* statements. In fact, most flow of control in machine language programs is typically by means of *goto* statements or similar constructs. Most high level languages include *goto* statements. Moreover, until very recently, most high level programming languages depended on the use of *goto* statements for much or even most flow of control.

Around 1960 a class of languages referred to as *structured programming languages* began to appear. A structured language is one that includes constructs for flow of control that allow for a systematic way to structure a program into meaningful subparts. Procedures and *while* statements are examples of such constructs. One of the earliest of these languages was ALGOL. Pascal is also a typical example of a structured language. These structuring constructs provided an alternative to the *goto* statement. Programs written with many *gotos* have a structure that is usually not apparent to the reader. Programs written without the *goto* can more easily exhibit a clear structure for the flow of control.

Although there are varying views on the details of how and when *gotos* should be used, some things about *goto* statements are clear. First of all, there is no absolute need for *gotos*. Any program that is written with *gotos* can be rewritten to do the same thing and to not include any *goto* statements. The question is whether *goto* statements enhance or detract from good programming style. Even in the less exact domain of style, some things are clear. A program that uses very many *gotos* is harder to read than a well-written program that uses very few *gotos* or no *goto* statements at all. A consensus has arisen that *goto* statements should be avoided.

When you are first learning to program, it is best to avoid *gotos* completely. Otherwise, it is difficult to learn that they are never needed and seldom even of any help. After you become proficient at programming, you may want to use an occasional *goto*, or you may agree with the school of thought that says they should never be used.

---

Situations in which *goto* statements may be reasonable are various sorts of exiting situations. When an error or other terminating condition is encountered, a *goto* can be used to jump directly to the end of a loop, a procedure, or an entire program. For example, a program can be designed to terminate on detection of an error, by the following scheme:

```

program Sample(input, output);
label 100;
var ...
...
begin {Program}
    ...
    if <error condition> then
        goto 100;
    ...
100: end. {Program}

```

In this scheme the label 100 labels the empty statement. This trick produces the equivalent of labeling the *end*.

Additional material on *goto* statements can be found in the references at the end of this appendix.

TURBO Pascal also has a statement called the *exit* statement that can be used for many of the same purposes as the *goto* statement. (The *exit* statement is not available in standard Pascal.) When the *exit* statement is executed, the procedure that is currently executing is terminated, and control returns to the program or procedure that called it. If *exit* is executed in the main body of the program, then the program terminates. In TURBO Pascal an alternative way to implement the above method of exiting on an error condition is the following:

*exit*

```

program Sample;
...
begin{Program}
    ...
    if <error condition> then
        exit
    ...
end. {Program}

```

In the previous paragraph, we outlined one use of the *exit* statement. That technique assumed that the *exit* statement was in the main body of the program and not in a procedure. If the *exit* statement is used in a procedure, then, when it is executed, the procedure will terminate, but the program will continue to run. The *exit* statement always exits from only one block. This action is often the desired action; other times it is not. To exit the entire program from within a procedure, you can use the TURBO Pascal *halt* statement. The *halt* statement always terminates the entire program.

*halt*



---

## References

- E.W. Dijkstra, "Goto Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3, March 1968, 147–148, 538, 541.
- D.E. Knuth, "Structured Programming with goto Statements," *Computing Surveys*, Vol. 6, No. 4, Dec. 1974, 261–298.
-

## Appendix 2

### *TURBO Pascal— More Integer Types*

In addition to the type `integer`, TURBO Pascal has four other data types for whole numbers. The one that is most likely to be of interest is the type `longint`. It allows whole number values which are much larger than the largest value of type `integer`. Figure A.2.1 lists the ranges of all the TURBO Pascal types for whole numbers along with the amount of storage used to store a value of each type. Aside from having different ranges of values, these types are used just like the type `integer`, but notice that the different types use differing amounts of storage. (A *byte* is eight bits of storage.)

For purposes of type compatibility, think of all these types as being subranges of the single type `longint`. Except for the obvious restriction that a variable cannot be assigned a value outside of its range, these types may be intermixed freely in expressions and assignment statements. The types interact with values of type `real` in the same way that the type `integer` does. The only major type compatibility pitfall has to do with variable parameters. Actual variable parameters must be of the exact same type as their corresponding formal variable parameters.

type	range	storage
byte	0..255	1 byte
shortint	-128..127	1 byte
integer	-32768..32767	2 bytes
word	0..65535	2 bytes
longint	-2147483648..2147483647	4 bytes

**Figure A.2.1**  
**TURBO Pascal**  
**types for whole**  
**numbers.**

## ***Appendix 3***

# ***Precedence of Operators***

Parentheses can be used to determine precedence; otherwise, operators are evaluated as follows:

First: *not*

Second: *\* / div mod and*

Third: *+ - or*

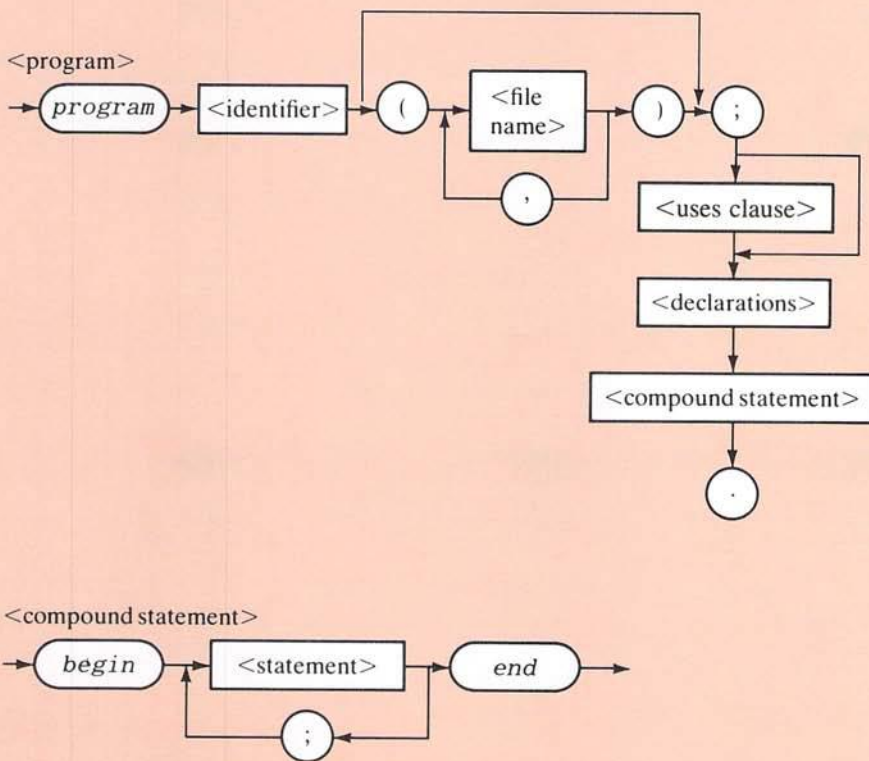
Fourth: *<= = >= > < <> in*

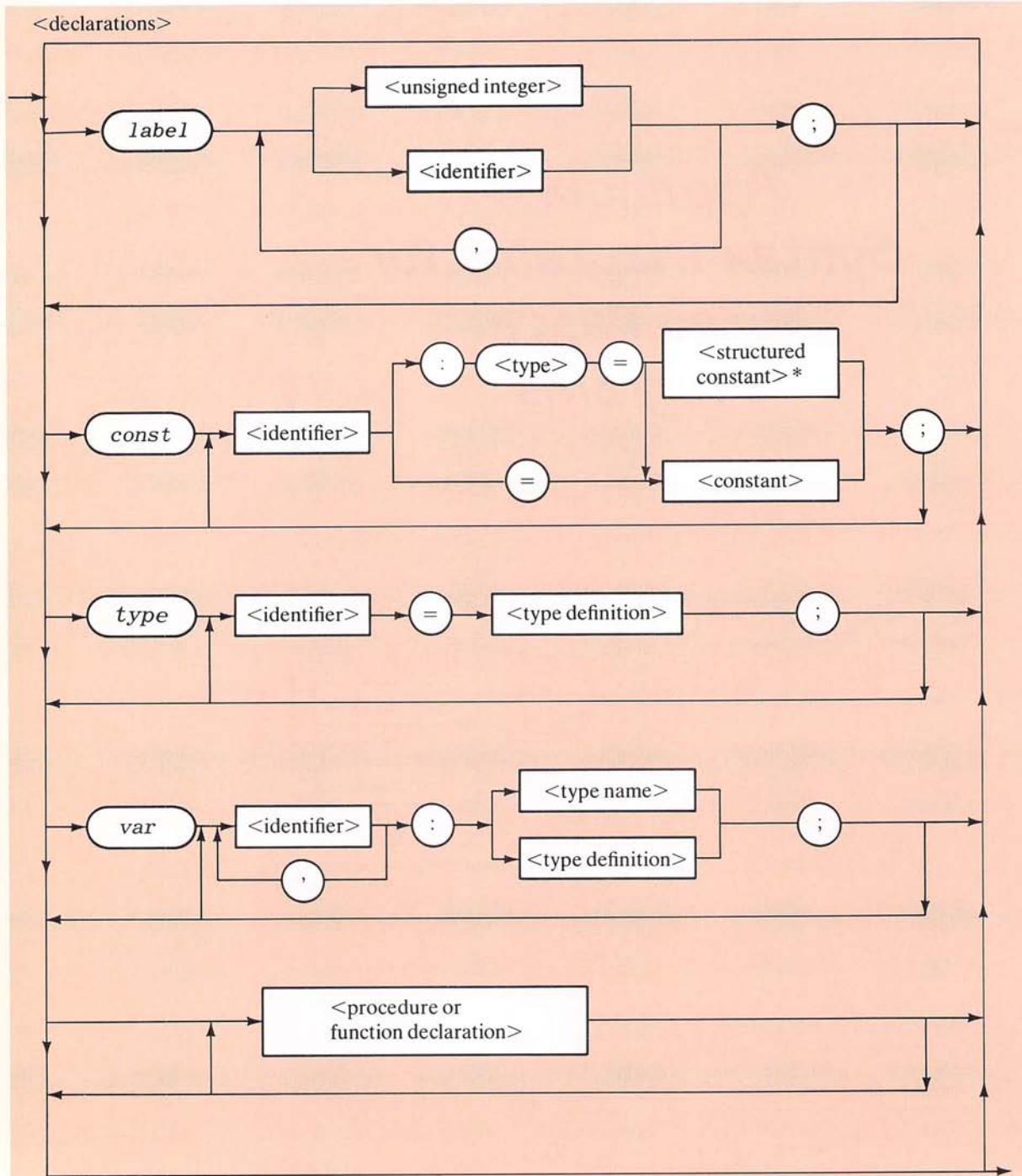
Operators in the same group are evaluated in their left to right order in the expression.



## Appendix 4

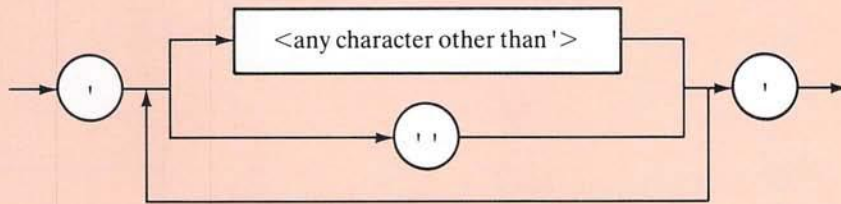
# Syntax Diagrams for TURBO Pascal Programs



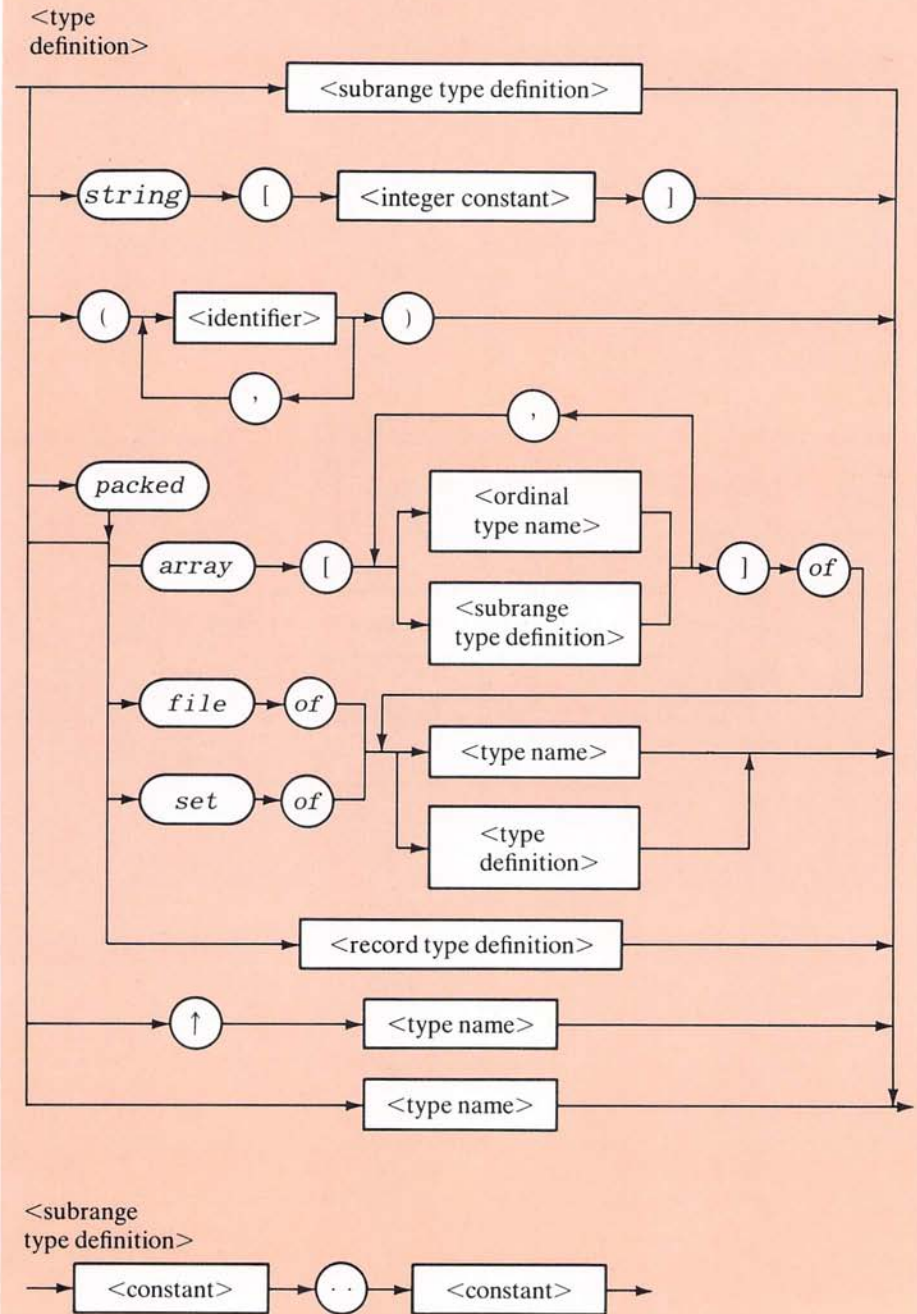


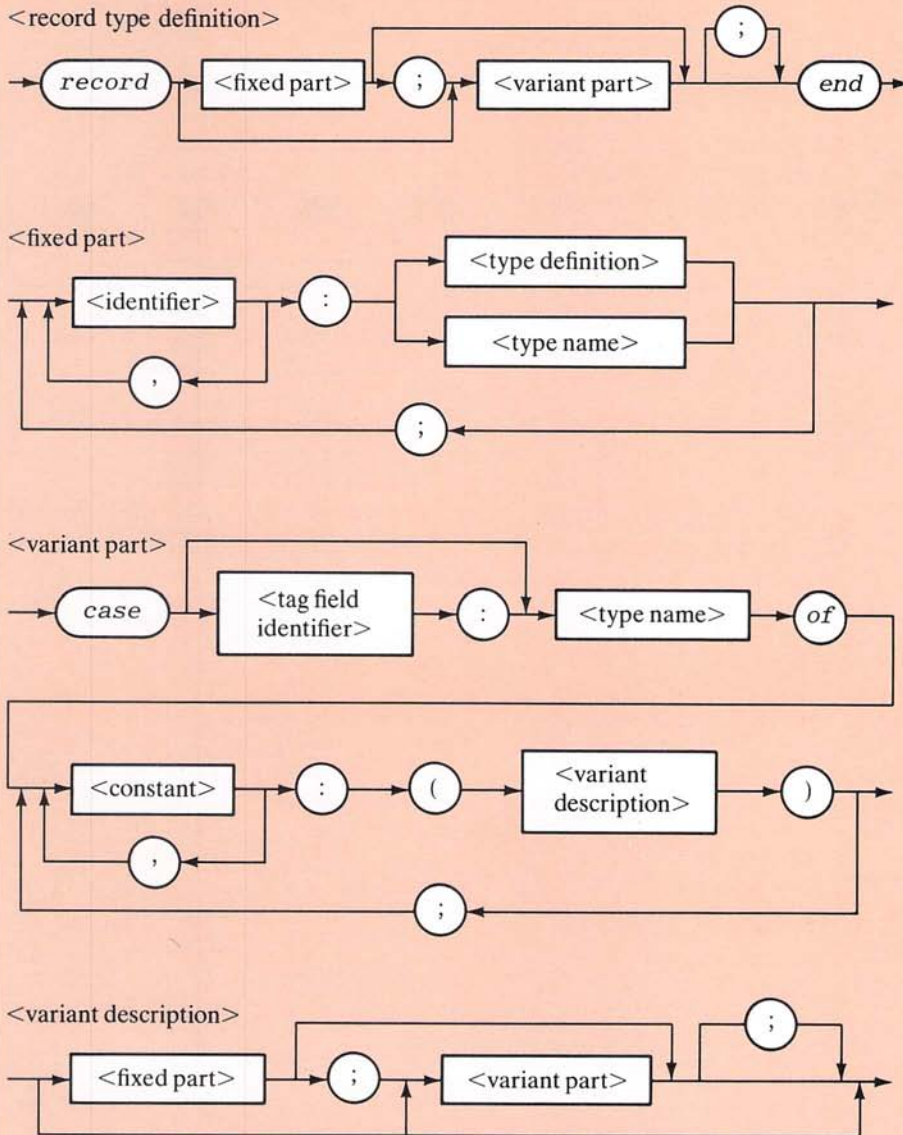
\* see chapters 10 and 11 for details.

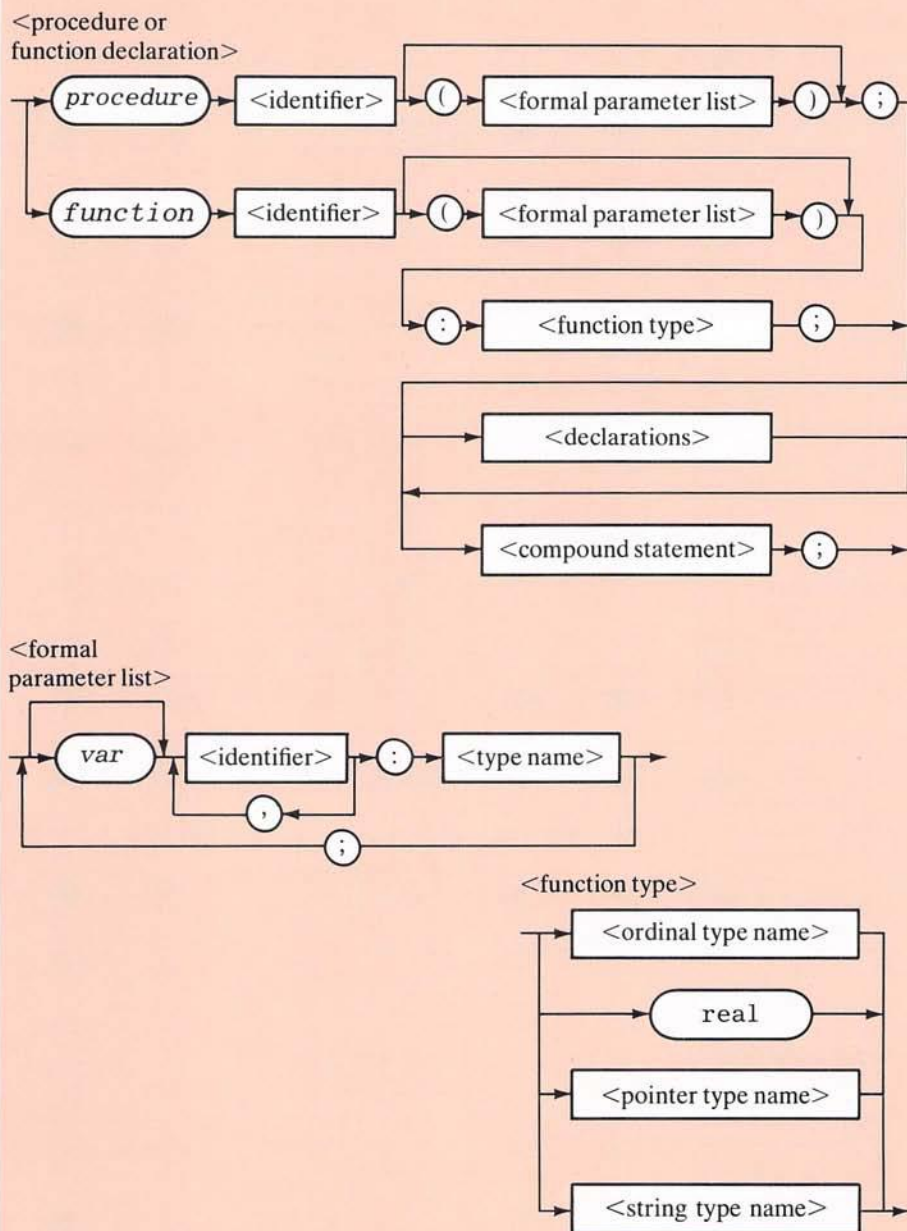
&lt;string constant&gt;



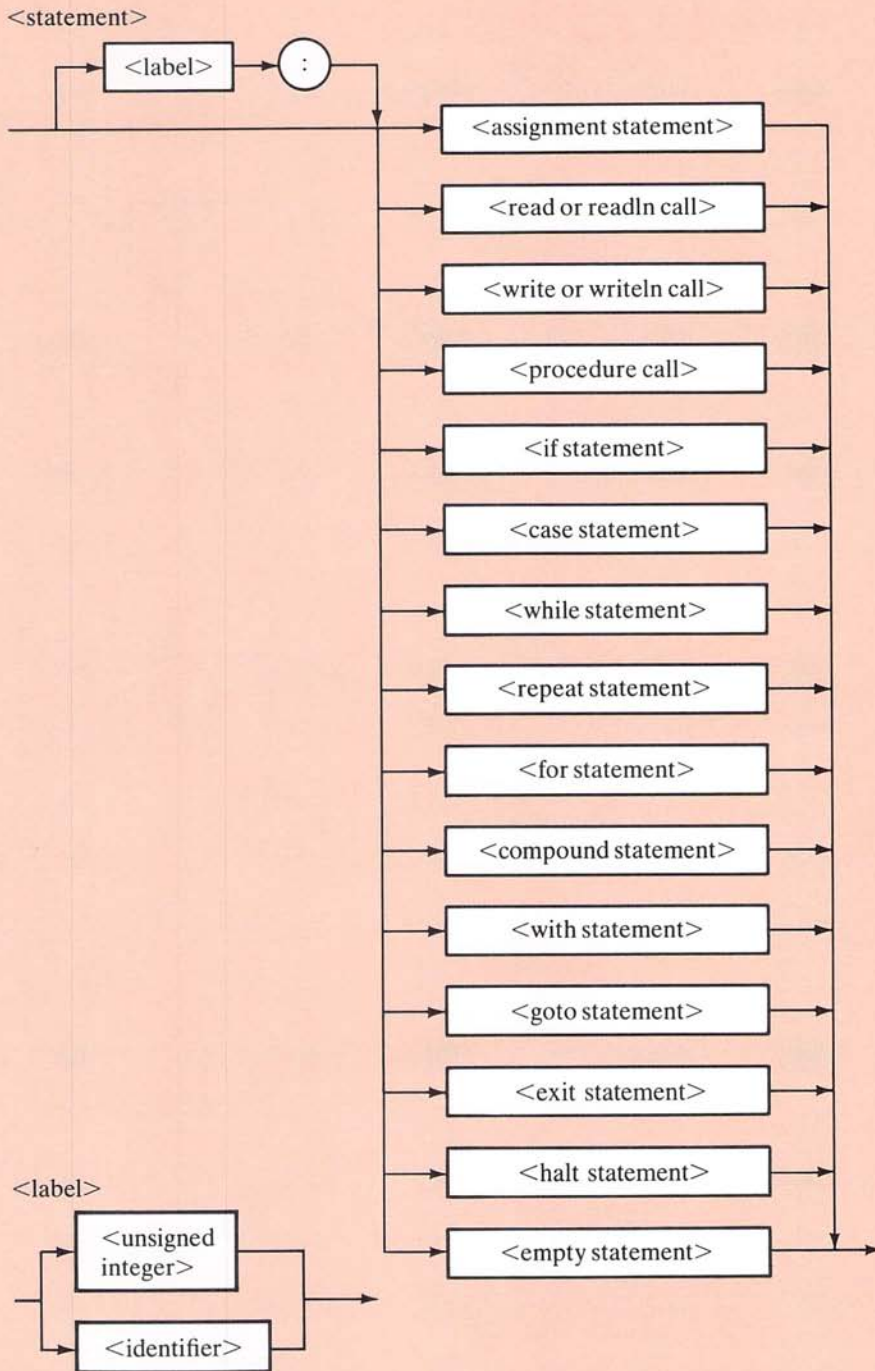


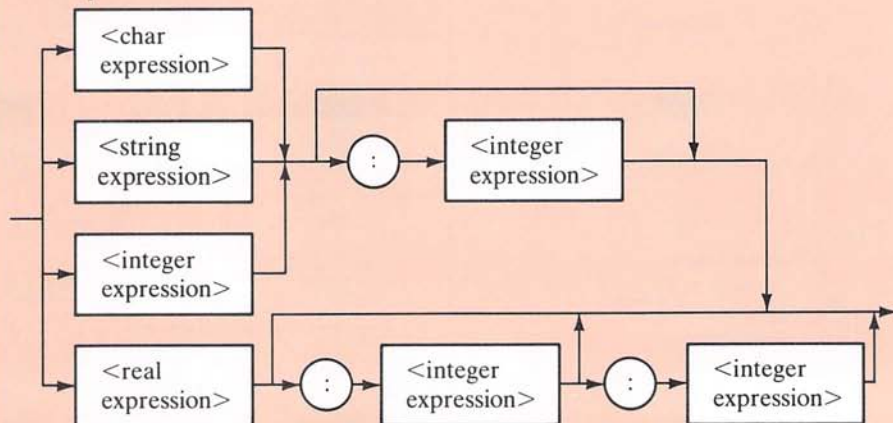
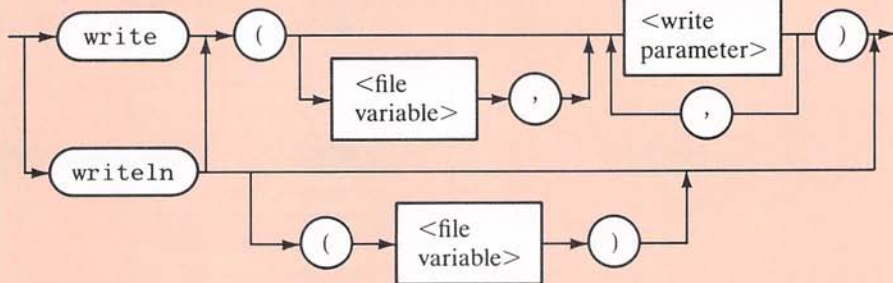
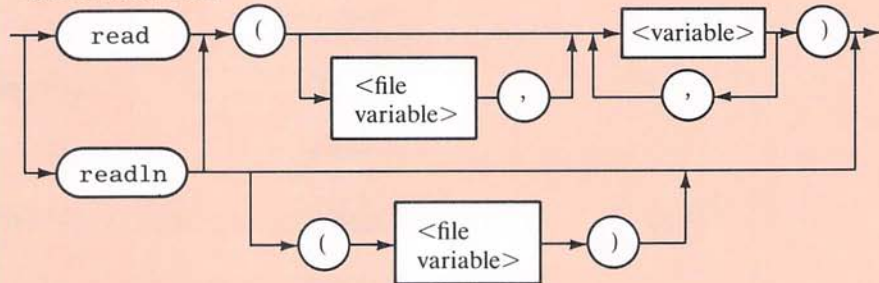




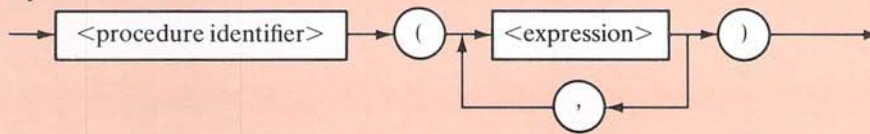




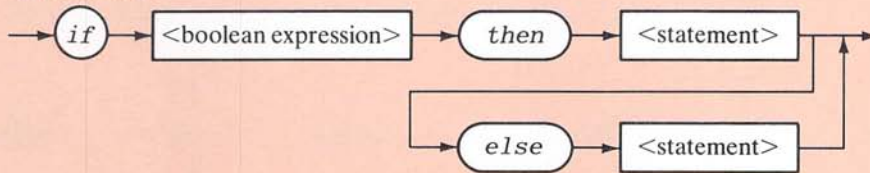




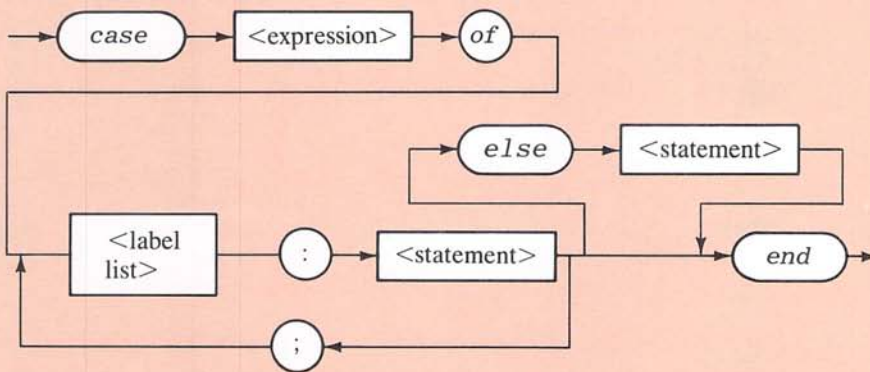
&lt;procedure call&gt;



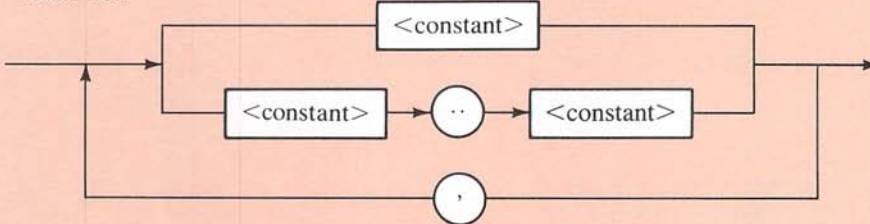
&lt;if statement&gt;



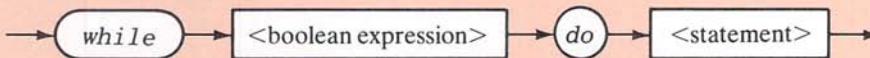
&lt;case statement&gt;



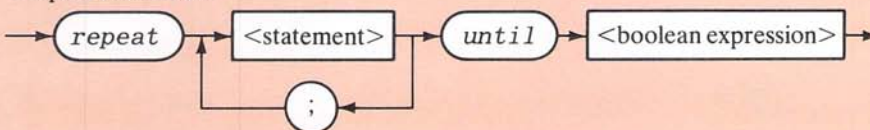
&lt;label list&gt;



&lt;while statement&gt;

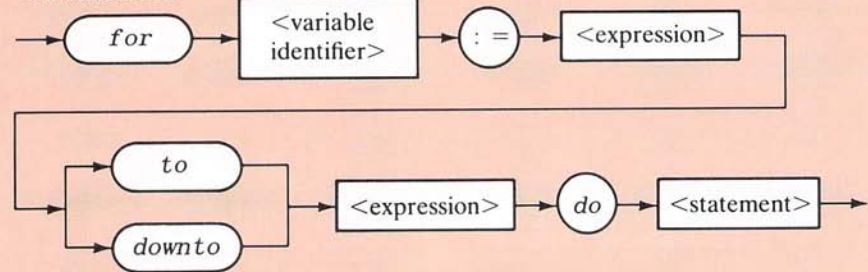


&lt;repeat statement&gt;

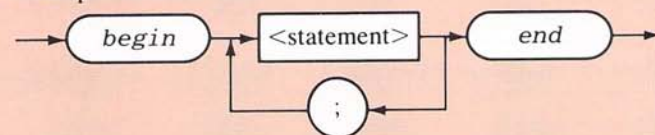




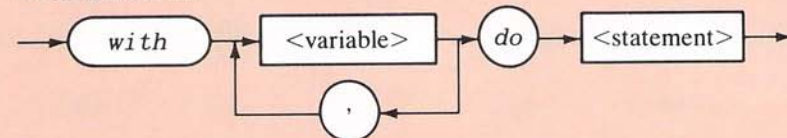
&lt;for statement&gt;



&lt;compound statement&gt;



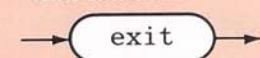
&lt;with statement&gt;



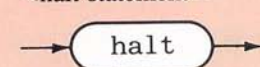
&lt;goto statement&gt;



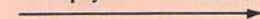
&lt;exit statement&gt;



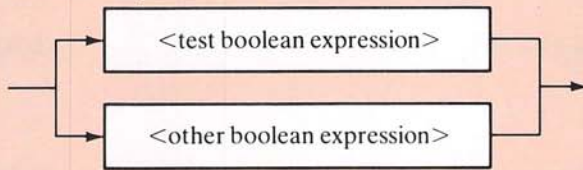
&lt;halt statement&gt;



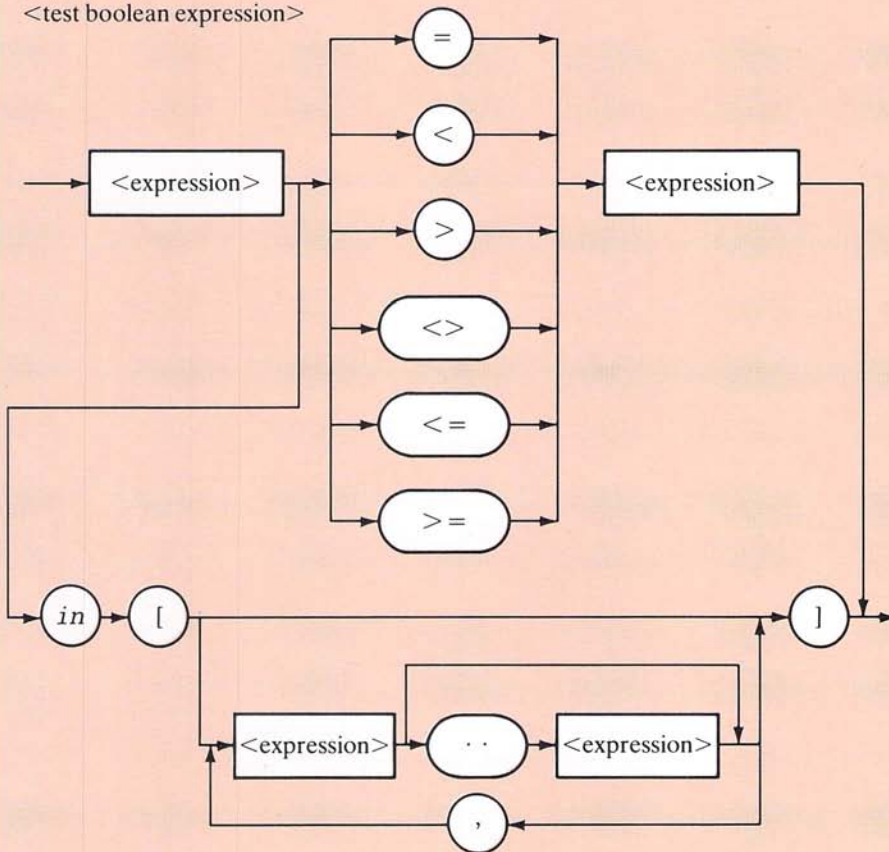
&lt;empty statement&gt;

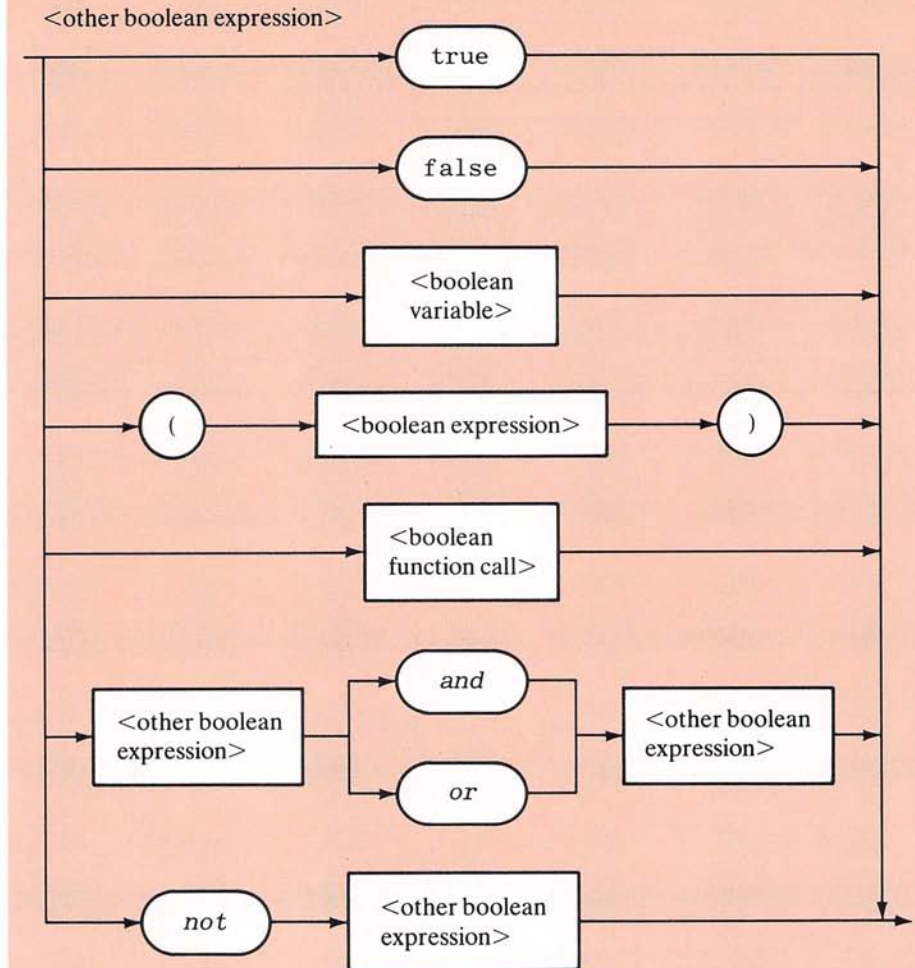


&lt;boolean expression&gt;



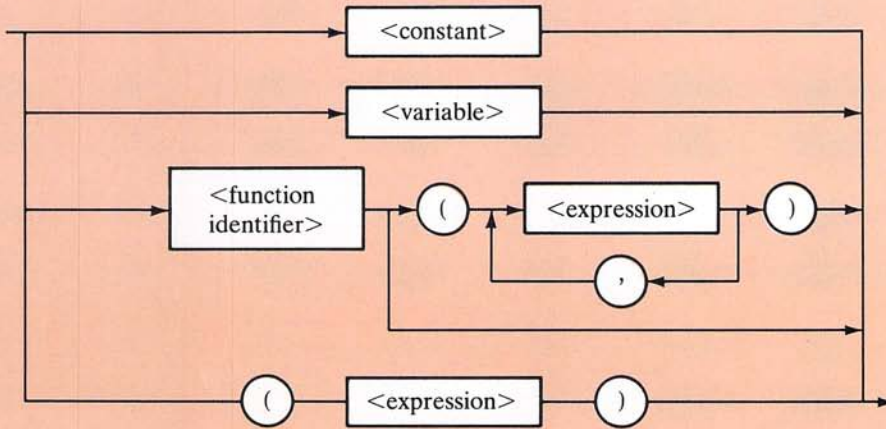
&lt;test boolean expression&gt;



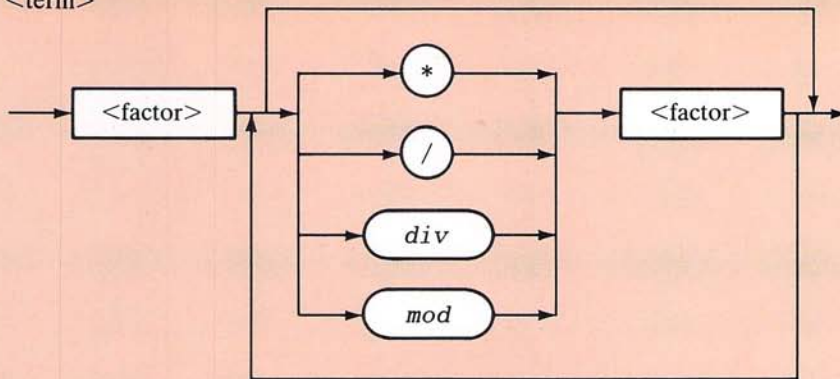




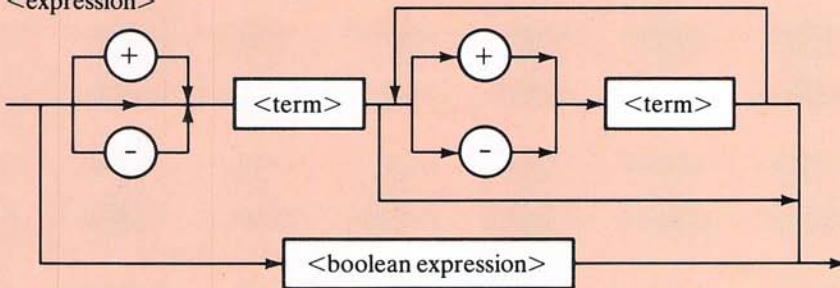
&lt;factor&gt;



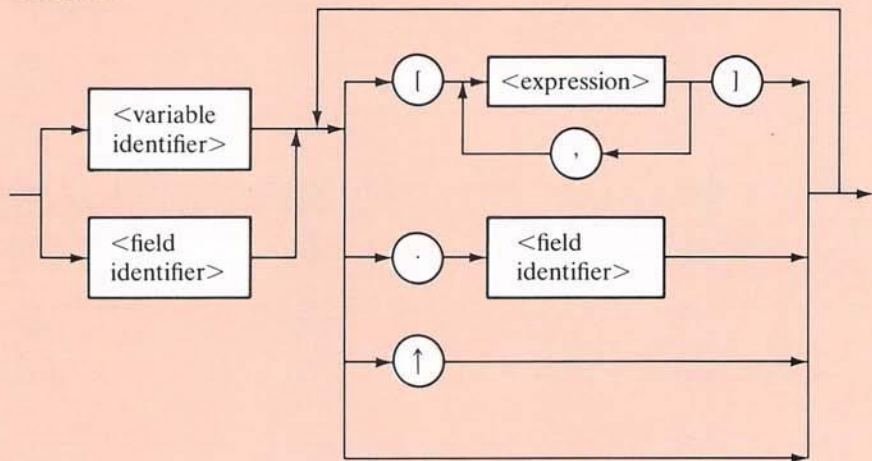
&lt;term&gt;



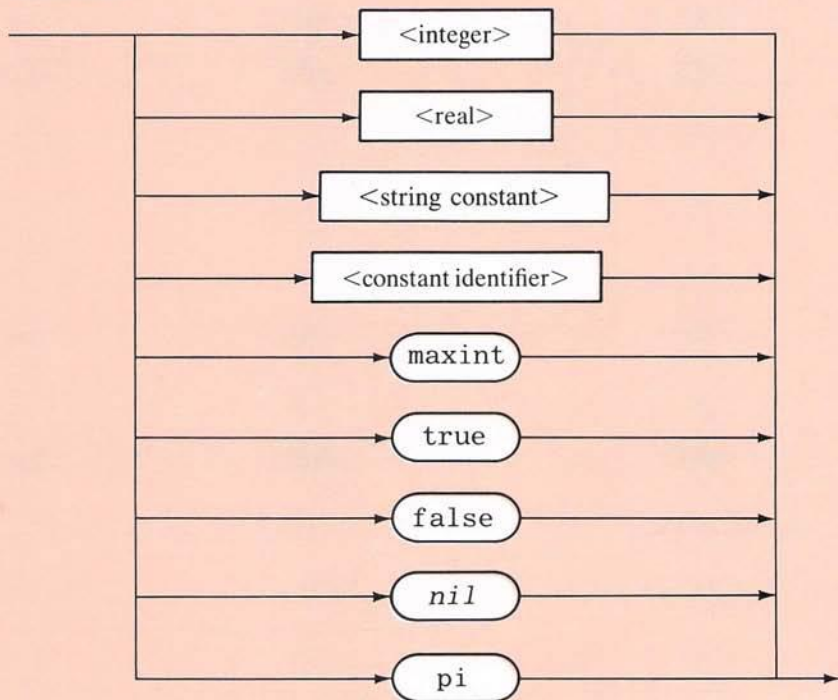
&lt;expression&gt;



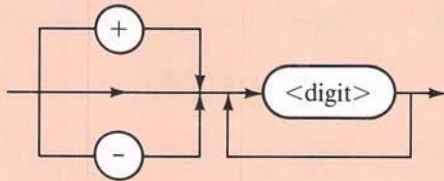
&lt;variable&gt;



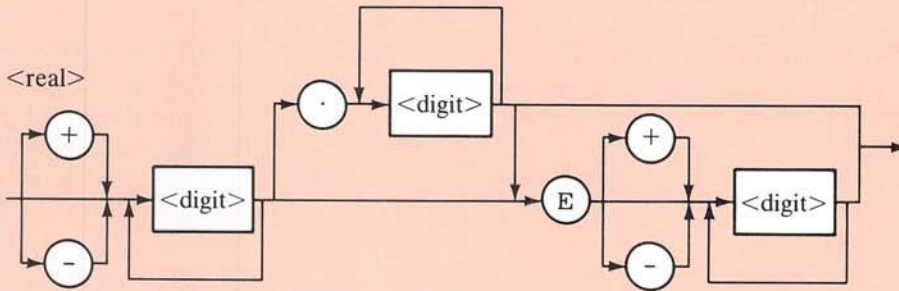
&lt;constant&gt;



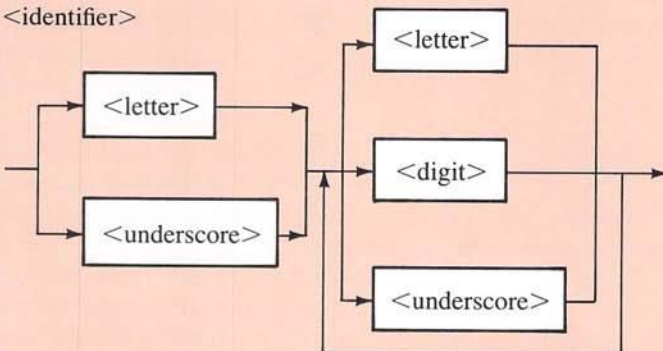
&lt;integer&gt;



&lt;real&gt;



&lt;identifier&gt;





## Appendix 5

# A Brief Introduction to the IBM PC and PC-DOS

Before reading this appendix, you should go back and read Chapter 1 in order to become familiar with basic computer terminology. You should also obtain access to an IBM PC as well as a floppy disk that contains the PC-DOS operating system. More powerful models of the IBM PC, such as the IBM XT and the IBM AT will, of course, serve the purpose. Most of what is presented here applies equally well if you are using a machine described as an IBM PC “look-alike” or if you are using the MS-DOS operating system.

---

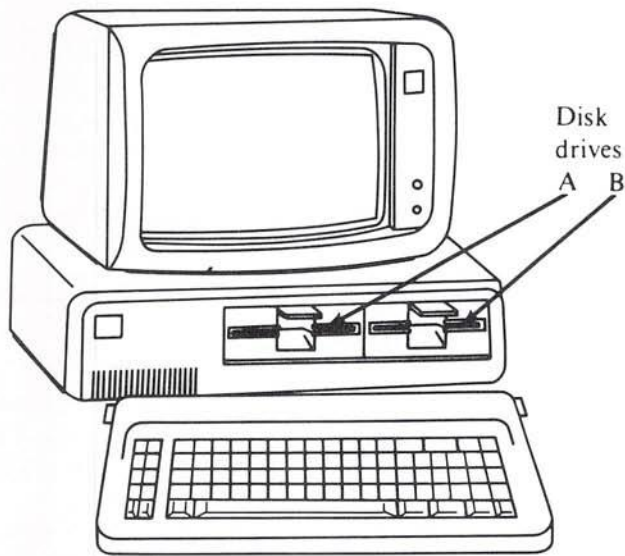
### Disks

DOS stands for “Disk Operating System,” which means that the computer must have a disk before it can do anything with this operating system. Virtually all machines have one or two *floppy disk drives*. In this appendix we assume you are using a machine with one or two floppy disk drives. Your computer may or may not also have a hard disk drive.

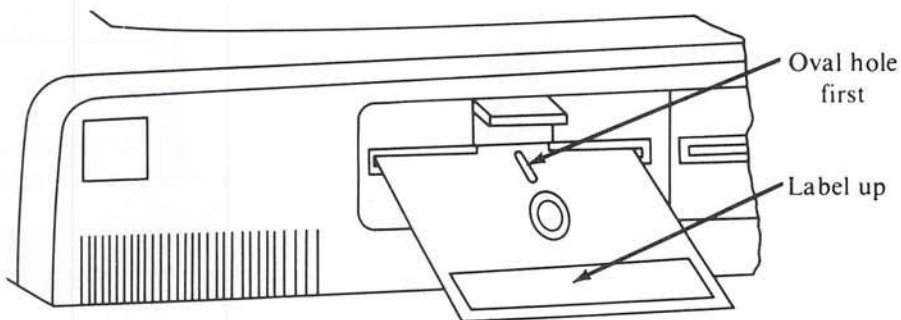
*drives*  
*A and B*

To start things off, insert a floppy disk with the operating system into the disk drive, close the door to the disk drive, and turn on your IBM PC (or similar machine). If you have two floppy disk drives, insert your disk in the main floppy disk drive, which is called *drive A*. (If you have a second floppy disk drive, it is called *drive B*.) On an IBM PC like the one in Figure A.5.1, drive A is the drive on the left. If you have two disk drives stacked one on top of the other, then drive A is normally the top drive. If you simply remove the disk from its protective envelope and insert it in the disk drive in the natural way, it should be oriented correctly. If the disk is turned around or is upside down, it will not function. When you insert the disk, the label should face up and the side with an oval hole in the outside shield should be inserted first. This insertion procedure is illustrated in Figure A.5.2.

---



**Figure A.5.1**  
**Floppy disk drives**  
**A and B on an**  
**IBM PC.**



**Figure A.5.2**  
**Inserting a floppy**  
**disk.**

## Booting—Getting Things Started

Once a disk with the operating system has been inserted into the main disk drive, you are ready to start using the DOS operating system. To get things started, you must *boot* the system. The term *boot* simply means getting the system started. The origin of the word is actually different, but you can think of it as “kicking the system with your boot to wake it up.” On most machines, all you need to do to boot the system is to turn the machine on. If it is already on, turn it off and wait a few seconds, then turn it back on. (If you are using an IBM PC, and it is already on and you do not want to turn it off, you can instead press the key labeled Del while holding down the two keys labeled Ctrl and Alt.)

---

## The Keyboard and Typing Details

Before going on, look over your computer’s keyboard and familiarize yourself with the location of a few keys. The keyboard for the IBM PC is illustrated in Figure A.5.3.

*return key*  
*(enter key)*

Be sure you can locate the *return key*. It is the “next line” key. It is also called the *enter key* and is likely to be labeled ↵.

*backspace*

Usually it is larger than the letter keys and is on the right-hand side of the keyboard.

Be sure you can also locate the *backspace key*. If you make a typing mistake, you can use this key to backspace over the mistake and retype it. On an IBM PC the backspace key is labeled with an arrow, like so ← and is located just above the return key.

If you misspell a command and do not discover the mistake before pressing the return key, the system will tell you it is a bad command and let you try again—provided you did not inadvertently type some other meaningful command.

When you are typing commands in DOS, there is no distinction between upper- and lowercase letters. You may use either or a mix. Hence, it is simplest to always use lowercase letters.

---

## Date and Time

Among other things, the computer works as a clock and calendar (while the machine is turned on). If you enter the correct time and date, the computer will keep track of them for you, labeling your files with correct date and time information.

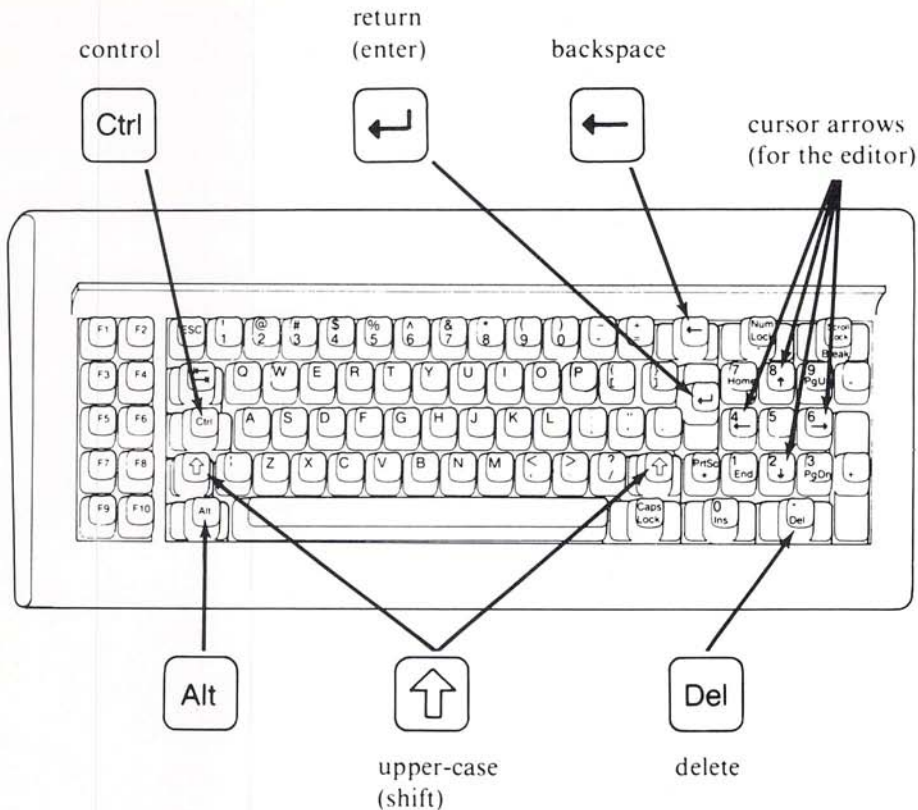
After the system boots, the computer will ask you to enter the date. Respond with three numbers separated by dashes. For example, if the date is January 12, 1989, then type the date as

**1-12-89**

If you do not wish to enter a date, simply push the return key until the computer stops asking (either once or twice). The computer will then use some leftover, usually incorrect date.

---





**Figure A.5.3**  
**Keyboard for IBM**  
**PC.**

Whether or not the computer receives a date, it will next request the time. Respond in 24-hour notation, like so:

**14:16**

for 2:16 P.M. Alternatively, just push the return key until it stops asking. In that case, it will use some leftover, usually incorrect time.

You can now use the computer as a calendar and clock. To see the current time, type `time` and then press the return key. The command `date` will display the date. The computer will ask for a new time and/or date. If you do not wish to reset the time or date, simply press the return key.

---

## The Prompt and the Default Drive

You are working in the DOS operating system, and within that system you will be provided with a *prompt*. The prompt will probably look like the following:

A>

*A and B*  
*prompts*

The prompt is given whenever the system is ready to accept DOS commands. When you then type in a command, the command will appear after this prompt. The letter A indicates that your *default* disk drive is drive A, which means that if you do not specify some other disk drive, the computer will always assume you mean disk drive A. For example, if you want to see the names of all the files you have on the floppy disk in disk drive A, you can type

```
dir a:
```

Do not forget the colon. (We will explain its significance later.) If you want to see what is on disk drive B, type

```
dir b:
```

If you simply type

```
dir
```

*changing  
the prompt*

the computer will assume you mean disk drive A. To change the default drive to drive B, type

```
b:
```

The prompt line will then change to

```
B>
```

If you then type

```
dir
```

it will list the files on the disk in drive B. To change the default drive back to A, type

```
a:
```

*stopping  
the screen*

If there are so many files that the list will not fit on the screen, you can request that the computer list one screenful at a time, waiting until you finish reading before proceeding to the next screenful. To do this, append /p to the end of the command. For example,

```
dir b: /p
```

will list the contents of disk drive B and pause after each screenful. The p stands for "pause." The slash is the way the system is told that a modification to the command is being given. Be sure to use the slash slanting in the direction shown. There are two slashes on the keyboard.

The command `dir` stands for "directory," and it tells the computer to list the names of all the files in the named directory. A *directory* is a collection of files, in this case the files on a floppy disk. The colon after a and b in the commands we just discussed are there to tell the system that we are referring to entire directories and not to single files. If you omit the colon in `a:`, the system will look for a file named a. We will say more about directories after we discuss files. (For now we are assuming that there are no relevant subdirectories on your floppy disk. If you do not know what a

---

subdirectory is and/or you do not know what we mean by “relevant,” then you can safely assume that there are no relevant subdirectories on your disk.)

If you have additional disk drives, they will be called drives C, D, and so forth. Any of these drives may be made the default drive by typing the name of the drive followed by a colon. For example, to make drive C the default drive, type

C:

Usually we will discuss only drives A and B, but the generalizations to more drives (if you have them) will normally be obvious. If you have only one floppy disk drive, it serves as both drive A and drive B. This is true even if you have a hard disk. If you have a hard disk and one floppy disk drive, then the floppy disk drive serves as both drive A and drive B. (In this case the hard disk will be drive C.)

*more or fewer  
disk drives*

---

## Ctrl-S and Ctrl-C—Stopping Things

(This section can be skipped on your first reading, but you should return to it at some point.)

Adding /p to the `dir` command is one way to stop the output from moving off the screen before you get a chance to read it. If you do not add /p to the `dir` command, you can still stop the list of files from going off the screen before you get a chance to read them. The command *control-S* will stop the screen. Whenever you want to stop the screen output on `dir` or almost any other command, you can type this command. To do so, first hold down the *control* key and then press the S key. The control key is usually labeled “Ctrl,” and so we will use the abbreviation *Ctrl-S* for control-S. `Ctrl-S` is typed in at the time that you want to stop the screen. To start the screen moving again, type `Ctrl-S` a second time.

If you wish to stop a command or program entirely, you can type *Ctrl-C*, done by holding down the control key and then pressing C. To remember this command, think of C as standing for “cancel.”

*stopping  
the screen*

---

## Files and File Names

A *file* is a collection of similar information grouped together as a unit, given a name, and stored on a disk (typically a floppy disk). For example, one file might contain a Pascal program. Another file might contain a love letter. Yet another file might contain the machine language code for the operating system. In the DOS operating system, files have two-part names. The first part is any string consisting of letters and digits and the special symbols `- _ ( ) { } % # & $ ! ~ ^ ' and ' .` The name may optionally have a second part called an *extension*, which consists of a period (pronounced “dot”) and a second string made up from the same list of symbols. The first name must be between one and eight characters long, and the second name must be zero to three characters long. When you type a name, you type it as one single word without any spaces, as in the following example:

FIRST.PAS



However, when the DOS system displays the file name on the screen, it may use one or more spaces in place of the dot. So the file FIRST.PAS may be listed by the computer as

```
FIRST PAS
```

*upper-  
and  
lowercase*

DOS does not distinguish between upper- and lowercase letters in file names. We will always use uppercase letters when we write file names; however, you may use either upper- or lowercase letters. To DOS, FIRST.PAS and first.pas are the exact same file name.

*type*

If you want to see what is in a file, you can use the command type. For example,

```
type FIRST.PAS
```

will cause the computer to look on the default disk for a file named FIRST.PAS and, if it is found, display it on the screen. Only certain kinds of files will reveal anything meaningful when displayed on the screen. Fortunately, files with Pascal programs in them are among the kinds that will.

*copy*

You can make copies of files. For example,

```
copy FIRST.PAS TEST.PAS
```

will make a second copy of the file FIRST.PAS and name it TEST.PAS. Both files are on the default disk. To copy a file from one floppy disk to another, you must specify the disk drives as well. (If it is a new disk, then the disk to receive the file must first be formatted. Formatting is discussed later on in this appendix.) For example,

```
copy a: FIRST.PAS b: FIRST.PAS
```

will make a copy of the file called FIRST.PAS on the disk in drive A, placing it on the disk in drive B and naming it FIRST.PAS on drive B as well. It is possible to use a different name for the file on disk B. Moreover, if no name is specified, it is given the same name as the original file had. So the following is equivalent to the command we just saw:

```
copy a: FIRST.PAS b:
```

If A is the default drive, you can even leave out the a: in this command.

*erase*

Unwanted files may be removed with the erase command. For example,

```
erase TEST.PAS
```

will eliminate the file TEST.PAS from the default disk. So if the prompt were

```
A>
```

and there were a file named TEST.PAS on the floppy disk in drive A, then it would be eliminated.

*rename*

There is also a command to rename files. For example,

```
rename FIRST.PAS PROG1.PAS
```

will change the name of the file FIRST.PAS to PROG1.PAS. All this takes place on the default disk.

---

As with people, one often refers to a file by its first name (part) alone. However, in almost all DOS commands you must give the full file name, including the extension, if there is one. If you mean TEST.PAS, do not write TEST alone.

When specifying a file name, you can use the asterisk as a *wild card* for part of the file name. Any string matches the asterisk. For example, the following command will delete all files on the default disk drive whose names start with the letter A and end with .PAS:

```
erase A*.PAS
```

The asterisk cannot be used in place of the dot (period), so to erase all the files on the default disk drive (a very drastic thing to do), the command is:

```
erase *.*
```

As a more useful example, the following will copy onto disk drive B all the files on disk drive A whose names end in .PAS:

```
copy a:*.PAS b:
```

---

## Backing Up a Floppy Disk

Numerous things can go wrong with the files stored on a floppy disk. You may inadvertently erase or change a file. Even if you make no mistakes in giving commands, the disk can be physically damaged by being left near heat or a magnetic field, or it can be damaged by being bent or otherwise abused. A floppy disk is a delicate item. To avoid losing the information stored on your floppy disk, use two floppy disks: one to work with and a second that contains copies of all your files. You can copy a single file using the copy command described earlier. A simpler thing to do is to make a copy of the entire disk. This is done with the diskcopy command.

To make an exact copy of your DOS system disk, insert the disk into disk drive A, and insert the disk to receive the copy into disk drive B. If the disk in drive B has any information on it, it will be lost and that disk will be made into an exact copy of the other floppy disk. After inserting the disks, make sure that the prompt line displays the letter A, indicating that the default drive is disk drive A, and then type the command

```
diskcopy a: b:
```

The computer will then ask you to insert the *source* disk (the one to be copied) into drive A and the *target* disk (the one to receive the copying) into drive B. Because they are already in the correct drives, you need only press any key and the copying will start.

If you have just one floppy disk drive, type the diskcopy command exactly as shown, including the a: and the b: . The computer will tell you when to insert the source and target disks. With one disk drive, you will have to change the disks several times, but the computer will tell you when to do so.

Actually, diskcopy is a program stored in a file. At first you will probably have only one disk, and diskcopy is on that disk. Hence, the above set of instructions

wild  
cards

diskcopy

should work fine. If the `diskcopy` program is not on the disk to be copied, proceed as follows: First place the disk with the file containing `diskcopy` into disk drive A (the file is called `DISKCOPY.COM`), then type the command as displayed in the previous paragraph. When the computer asks you to insert the disks to be worked on, insert the two disks as directed and then press any (single) key.

*diskcomp*

When you are using `diskcopy`, it is a good idea to test to make sure the copying was performed successfully. The command `diskcomp` should be used to compare the two disks to see if they are identical. The command

```
diskcomp a: b:
```

will compare the two disks. (The command is the same whether you have one or two disk drives.) If the comparison is not successful, redo the `diskcopy`.

*chkdsk*

It is also a good idea to check disks from time to time to see that they are not damaged and to see how much storage is available on them. This can be done with the command

```
chkdsk
```

which will check the disk in the default drive for some types of problems and will also tell you about available storage on the disk.

As with `diskcopy`, the commands `diskcomp` and `chkdsk` are programs contained in files. In these cases, the files are called `DISKCOMP.COM` and `CHKDSK.COM`, respectively. When one of these commands is entered, a disk that contains the file for that command must be in the default disk drive.

---

## Formatting Disks

Before you can start copying files to a brand new disk, you must format it. (The one exception to this rule is when you use `diskcopy`, which does not require that you format the disk.) Formatting prepares the disk to receive files. Formatting a disk for files is analogous to putting in the streets before you build the houses in a new housing development.

To format a disk, you use the program `format`, called `FORMAT.COM` in the directory listing. To format a disk, place another disk that contains the file `FORMAT.COM` in disk drive A, and place the disk to be formatted in drive B. Be sure the prompt line is

```
A>
```

and then type the command

```
format b:
```

If you have only one floppy disk drive, use this same command (including the `b:`). The computer will tell you what to do with the disk to be formatted.

Some occasions other than having a brand new disk also call for formatting a disk. Any time you wish to reuse a disk and discard the information on it, you should reformat the disk. This will not only ensure that it is formatted correctly but will also erase

---



all the files on the disk. A disk that has been formatted for a different operating system will also need to be reformatted before it is used. Formatting can even rescue a disk from some kinds of malfunctioning. A disk that produces errors can sometimes be made usable again simply by reformatting it, but remember to save any files you want onto another disk before you do the formatting.

A disk that is formatted as we described above can be used to store files. Once the system has been booted, i.e., once the system is running, the disk can have files read from it and/or written to it. However, the disk cannot be used to boot the system unless it is a *system disk*. To make a disk into a system disk, append /s to the end of the format command as follows:

```
format b: /s
```

Any disk formatted in this way can be used to boot the system. There is a penalty, however, for making a disk into a system disk. A system disk contains a copy of the system, and so it will have less space available for your files.

*bootable  
disks*

---

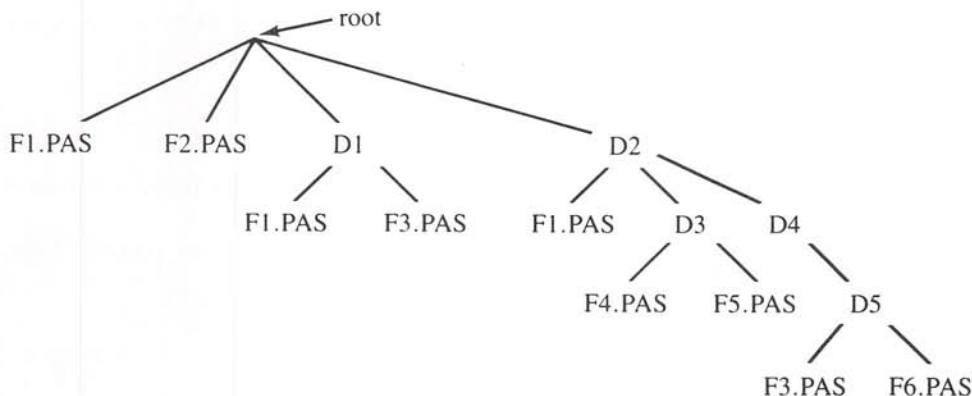
## Directories

(This section may be delayed or omitted without a loss of continuity.)

Files may be grouped together into named units called *directories*. In the simplest case, there is one directory per floppy disk that contains all the files on the floppy disk. One can also create subdirectories, and these subdirectories may also have subdirectories, and so forth. These subdirectories are a way of organizing the files into natural groups.

DOS directories are said to be *tree structured*, which means there is a hierarchy of files. At the top (i.e., for the entire disk) there is a main directory, called the *root directory*, which may contain both files and subdirectories. Each of these subdirectories may contain files as well as their own subdirectories, which may in turn contain files and subdirectories, and so on. For example, Figure A.5.4 shows one possible arrangement of directories and files. To see why it is called *tree* structured, turn the page upside down.

*tree  
structure*



**Figure A.5.4**  
Directories and  
Files.

Once there are subdirectories, naming of files can get a bit more complicated. Notice that two different files can have the same name, provided they are in different subdirectories. The floppy disk described in Figure A.5.4 has three files named F1.PAS. How can you tell which file we mean when we give a command like the following?

```
erase F1.PAS
```

*changing  
directories*

The answer is as follows. You are always considered to be in some directory or subdirectory or sub-subdirectory of the default disk drive. When you start, you are in the root directory, so this command would eliminate the file that is just below the root. You can change your location with the command `cd`. (An alternative spelling of this command is `chdir`.) To get to the directory D1, the command is

```
cd D1
```

If this command had preceded the `erase` command, then the file F1.PAS in directory D1 is the one that would have been removed.

To go up one level in the tree, the command uses two dots (periods) in place of a directory name. Hence,

```
cd ..
```

moves you up one level, in this example returning you to the root directory. This is the normal way to move up the (upside-down) tree. Similarly, you normally move down one level at a time. To get to the directory D3, starting from the root directory, you use two commands:

```
cd D2 <return key>  
cd D3 <return key>
```

The second command alone will not work. It is not that easy to “jump over” directories.

*jump to  
root*

However, there is a special command to jump directly to the root directory. The command

```
cd \
```

will move you from any directory to the root directory.

The command `dir` and virtually all other commands apply to the directory you are currently working in.

*mkdir*

To create a directory, use `mkdir`. The command

```
mkdir SAM
```

will create a subdirectory of the directory you are currently working in and will name it SAM.

*rmdir*

To eliminate a directory, use `rmdir`. The following command gets rid of this newly created subdirectory:

```
rmdir SAM
```

---

If you forget where you are and want to know what directory you are in, type

```
cd
```

without any directory name.

If you are in a particular directory, then you can easily name the files in that directory. If you wish to name a file in some other directory (without changing your directory), then you must specify what directory the file is in. One way to do this is to use a *full path name*. A full path name contains a list of the directories you would pass through in going from the root directory to the file. The directory names are separated by slashes. The path name ends with the name of the file and begins with an extra slash (to stand for the root directory). For example, the full path name for the only file named F4.PAS in Figure A.5.4 is

*path  
names*

```
\D2\D3\F4.PAS
```

This full path name can be used to name the file and will have the same meaning no matter what directory you are in. To delete this file, the following command will work whether you are in the root directory, directory D1, or any other directory:

```
erase \D2\D3\F4.PAS
```

Although two different files in two different directories may have the same name, they never have the same full path name. Figure A.5.4 shows two files named F3.PAS. Their full path names are

```
\D1\F3.PAS and  
\D2\D4\D5\F3.PAS
```

Path names are also used to name directories. The full path name for the directory D3 in Figure A.5.4 is

```
\D2\D3
```

If you are manipulating files that are in subdirectories of the directory you are in (or subsubdirectories or anywhere “below” your current directory), then you can abbreviate the full path name to a *relative path name*. If the path name does not start with a slash, then the system assumes that the first part of the path name contains the directories on the path from the root directory to your current directory. For example, if you are in the directory \D2\D4, then the relative path name

*relative  
path names*

```
D5\F6.PAS
```

is an abbreviation for the full path name

```
\D2\D4\D5\F6.PAS
```

The meaning of this abbreviation depends on what directory you are in. If you want names that mean the same no matter what directory you are in, then you must use full path names.



## *Appendix 6*

# *Summary of DOS Commands*

In the following descriptions, we assume that there are two disk drives, drive A and drive B. If there are more drives (drive C, drive D, etc.), then the names c, d, etc. may be substituted for a or b.

**date**

Returns the date and allows you to reset the date.

**time**

Returns the time and allows you to reset the time.

**a:**

Makes A the default disk drive.

**b:**

Makes B the default disk drive.

**dir**

Lists the files in the default directory. Follow with a: or b: to request the files on disk drive A or B, respectively. To produce a pause after each screenful, append /p to the end.

**type <file name>**

Displays the file named <file name> on the screen.

**copy <name of file to be copied> <new name>**

Copies named file to produce a second copy with the <new name>. All this takes place on the default drive unless disk drives are specified. To specify a disk drive, add a: or b: to the front of one or both file names. For example,

**copy a: FIRST.PAS b: SPECIAL.PAS**

**erase <file name>**

Removes the named file from the disk in the default drive. To specify another drive, add a: or b: to the front of the <file name>.

**rename <old name> <new name>**

Changes the name of the specified file on the disk in the default drive.

**format b:**

Formats the disk in disk drive B. a: may be used instead of b: to format the disk in drive A. To obtain a disk that contains the "system" and so can be booted by itself, append /s.

---

**diskcopy a: b:**

Copies the contents of the disk in drive A onto the disk in drive B. Afterwards, the disk in B is identical to the one in A. The roles of a: and b: may be reversed.

**diskcomp a: b:**

Compares the disks in drive A and B to see if they have the same contents.

**chkdsk**

Checks the disk in the default drive for certain problems and reports the available storage on the disk.

**cd <argument>**

(alternative spelling **chdir <argument>**)

Changes directory. The effect of this command depends on the <argument>.

**cd (no <argument>)**

Tells your current directory location.

**cd \**

Makes the root directory your directory location.

**cd ..**

Moves you to the directory directly above where you are.

**cd <directory name>**

Moves you to the named directory.

**mkdir <directory name>**

Makes a new directory with the specified name as a subdirectory of your current directory.

**rmdir <directory name>**

Removes the named subdirectory of the current directory.

**Ctrl-S**

Stops screen output. Output can usually be resumed by again typing Ctrl-S.

**Ctrl-C**

The *break* ("cancel") command. Tells the computer to stop what it is doing.

---

# ***Appendix 7***

## ***A Quick***

### ***Introduction to the***

#### ***TURBO Menu***

##### ***Environment***

###### ***with Emphasis***

###### ***on the Editor***

Before reading this introduction, you should first go back and read at least Chapter 1, which gives a general introduction to computers and computer terminology. If you are using the DOS operating system and are not familiar with that system, then you should also read Appendix 5 before going on. If you are using an IBM PC, XT, AT, PS/2, or similar machine to run TURBO Pascal, then Appendix 5 undoubtedly applies to you.

The TURBO Pascal environment includes an editor and facilities for manipulating files as well as the TURBO Pascal compiler and a number of other utility programs. This appendix will take you through the writing and running of a simple Pascal program using the TURBO Pascal environment. Follow it when you are at the computer keyboard and you can actually type in the various commands. After that, you can go on to practice with the editor. When you feel somewhat comfortable with the editor, you can then begin to write TURBO Pascal programs. For now, do not worry much about the Pascal program we will give you to type in. Simply type it in as instructed. When learning to program a practical rule is: first learn to type, then learn to program. Pascal, including TURBO Pascal, is covered in the main body of this book.

*control  
characters*

*Alt  
key*

Before using the TURBO environment you should first familiarize yourself with the keyboard. It is configured like an ordinary typewriter keyboard but has a few additional keys. As with an ordinary typewriter keyboard, there are both upper- and lowercase letters. If you simply type a letter, that produces the lowercase versions. To type the uppercase version, you hold down the shift key and press the letter key, just as you

---



would on a typewriter. (The shift key is probably labeled either with an arrow pointing up or with the word “shift.”) This shift key doubles the number of available letters. There are 26 letters in the alphabet, but if we consider upper- and lowercase letters to be different symbols, then there are 52 letter symbols. The shift key gives us an extra 26 letter symbols, but that is not sufficient for our purposes. We will need still more versions of each letter. The computer keyboard provides two additional keys that work just like the shift key to produce more versions of each letter. One of these keys is labeled “Ctrl,” and is called the *control* key. The other is labeled “Alt,” and is usually simply called the *Alt* key. “Alt” is undoubtedly an abbreviation for “alternate,” although few people call it the “alternate key.” Both the control key and the Alt key are usually near the left end of the keyboard. These two extra keys give us four versions of each letter, the lowercase version, the uppercase version, the control version, and the Alt version. For example, to type the control version of the letter K hold down the control key and then press the key for the letter K. We will use the notation *Ctrl* to denote the control versions of letters. So, for example, *Ctrl-K* means “control K” formed by depressing the K key while holding down the control key. (We use uppercase letters in expressions like *Ctrl-K*. However, you do not hold down the shift, i.e., “uppercase key,” when typing control or Alt versions of letters.) Similarly, we will use the notation *Alt* to denote Alt versions of letters. So, for example, *Alt-K* means the Alt version of the letter K. Alt versions of letters are formed in the same way as the control version except that the Alt key is used in place of the control key.

Use of the control key and Alt key is not limited to the letter keys. You can type a control version or an Alt version of any key. You could type a *Ctrl-7* for example or a *Alt-5*. Since their use is not limited to letters, one seldom uses the term “control letter” or “control version of a letter.” Instead, one uses the term *control character*. Similarly, you might hear the term *Alt character*.

Before we conclude this tour of the keyboard, you should locate the *function keys*. At the left end of your keyboard or on the top row of your keyboard (or on some machines in some other place on the keyboard) you will find a collection of keys labeled “F1”, “F2”, etc. These keys are called *function keys*. As we will see, a number of TURBO environment commands can be given using these keys. These keys are often used in their control or Alt version. For example, *Alt-F5* is formed by pressing the F5 key while holding down the Alt key. Now that you understand the keyboard, it is time to start the TURBO environment and write your first TURBO Pascal program.

Before you can use the TURBO environment you may need to insert into your computer a floppy disk containing the TURBO environment, and turn the computer on or otherwise start the system. You may also need to enter the date and time as described in Appendix 5. After that, all you need do in order to start the TURBO Pascal environment is to type the word

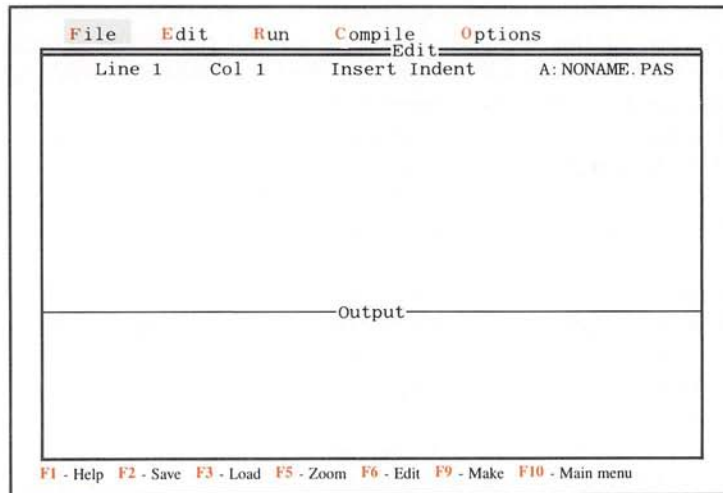
turbo

followed by pressing the return key. The *return* key is the “next line” key. It is also frequently called the *enter* key. Most of the TURBO commands we will discuss next are not ended by pressing return, so you should not do so automatically.

In response to your typing *turbo*, the system will display a copyright notice. (If you do not get a copyright notice, but the screen looks like Figure A.7.1 or A.7.3, do

*function  
keys*

*starting  
TURBO*



**Figure A.7.1**  
The version 4.0  
main screen.

not be concerned.) The copyright notice will tell you whether you are using version 4.0 or version 5.0 of the TURBO Pascal environment. Since there are differences between the two versions we have occasionally included separate sections for each version. If you are using version 4.0, you should read the sections labeled “Version 4.0” and skip the sections labeled “Version 5.0.” If you are using version 5.0, you should do the opposite. Sections without a version number apply equally to both versions and should be read no matter which version you are using.

## Version 4.0

### The Menu Environment

#### *main menu*

If you have started the TURBO environment so that the version 4.0 copyright notice is displayed, then you are ready to start learning how to use the environment. If you press the space bar, the copyright notice will disappear and you will be left with the *main screen* display for the TURBO environment as shown in Figure A.7.1. This main screen contains two lists of commands, one on the top line of the screen and one on the bottom line of the screen. These are lists of commands available to you. For now we will ignore the bottom line except for the last entry “F10–Main menu.” The F10 refers to the function key labeled F10. As you proceed to use the TURBO system, the main screen will often become covered with other displays. You can always clear the main screen by pressing the F10 key. Anytime you become confused and wish to start over again, you can press the F10 key and that will return you to the starting point so that you can enter the commands listed on the main screen.

For now let us concentrate on the first line of the main screen display. This first line is called the *main menu* and contains the following entries:

## File Edit Run Compile Options

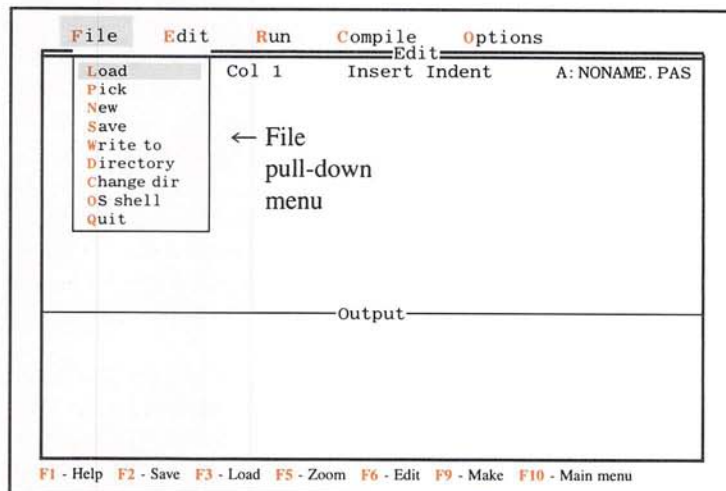
Each of these entries is the name of a command. If the main menu is active, then you can give any of these commands by typing the first letter of the command: F for File, E for Edit, and so forth. Typing any one of these letters will either cause the TURBO system to do something for you (such as running a program) or else it will produce a submenu displaying other commands. In this appendix we will use uppercase letters for these one letter commands in order to make them stand out from the rest of the text. You may use either uppercase or lowercase letters. When you start the TURBO environment, the main menu is active and so these commands are available to you. You can always make the main menu active by pressing the F10 key. If you wish to use commands on the main menu and are not sure if it is active, then simply press the F10 key. It will not cause any problems to press it an extra time.

Make sure the main menu is active by pressing the F10 key, and then try pressing the F key. (Lowercase f is fine.) That will produce the File submenu as shown in Figure A.7.2. A submenu display such as the File submenu shown in Figure A.7.2 is called a *window*. Since it seems to come down from the top of the screen it is often referred to as a *pull-down window* or *pull-down menu*. Making the window appear is called *opening* the window. When a pull-down menu is open (i.e., displayed), the main menu becomes inactive and the pull-down menu becomes the active menu. You can then give any of the commands shown in the pull-down window. There are other pull-down menus, but for now let us concentrate on the File pull-down menu. Before going on make sure that the File pull-down menu is active; in other words, make sure the screen display looks like Figure A.7.2. (Pressing F10 followed by F will ensure this.)

As with the main menu, the commands in the File pull-down menu can be given by typing the first letter of the command. For example, if you were to press the Q key for Quit, that would cause you to leave the TURBO environment. This is the command you give when you are finished with the TURBO environment. As a sample command,

*pull-down  
menus*

*File menu*



**Figure A.7.2**  
**Version 4.0: the**  
**File menu.**



try pressing the D key to choose the **D**irectory command. This will cause yet another window to appear on the screen. It will look something like

Enter Mask
*.*

The **D**irectory command will list the names of the files in your current directory, but before it lists them it displays this window as a way of asking you whether you want to see all the names or just some of the names. If you simply press the return key, you will see the names of all the files in your current directory. Try it. The small window will be replaced with a larger window which contains the names of all the files in your directory.

*closing  
windows*

Next try typing one of the other commands on the main menu such as E for **E**dit or R for **R**un. Nothing will happen. The only commands that are available are the ones shown in the last window that was opened. At this point, the last window contains no commands, only file names. In order to execute commands on a menu you must get rid of any windows that are “on top” of that menu. To get rid of a window, you press the *escape* key. (Look for a key labeled “Esc” or something similar. Try the upper left-hand portion of the keyboard.) When you press the escape key, the last menu that you produced will disappear. Try pressing the escape key. That should get rid of the window with the list of file names. Getting rid of a window is called *closing* the window. If you press the escape key only once, then the window with the list of files will be closed, but you will still have the **F**ile pull-down menu displayed and so can issue commands on that menu, such as typing D for **D**irectory or Q for **Q**uit, or typing the first letter of any other command on that menu. However, if you want to give one of the commands on the main menu, such as E for **E**dit or R for **R**un, then you must first close the window containing the **F**ile menu. To close this window, press the escape key one more time.

*F10 key*

If you have not already done so, try closing the **F**ile window by pressing the escape key. For more practice try opening and closing the **C**ompile menu by pressing the C key and then pressing the escape key. Next try opening and closing the **O**ptions menu by pressing the O key and then the escape key. Finally, you might repeat the exercise of opening the **F**ile menu and then pressing the D key to choose the **D**irectory command on that pull-down menu. After you produce the list of files on the screen, you have two methods available for getting back to the main menu. You can press the escape key twice to close the two small windows or you can simply press the F10 key once. Try doing the complete exercise of starting at the main menu, displaying the list of files and then returning to the main menu. Do it two times, once return by pressing the escape key twice and once return by the shortcut of pressing the F10 key once. We will return to explain these windows and menus in more detail, but first we will use the editor to write a Pascal program.

*hotkeys*

There are some short cuts that simplify the processes of closing and opening menu windows. If you type the Alt version of the first letter of any command on the main menu, that will execute that command no matter what window is active. For example, pressing Alt-C will activate the **C**ompile pull-down menu; pressing Alt-F will activate the **F**ile pull-down menu; pressing Alt-E will activate the editor by execut-

ing the **Edit** command. These are called *hotkeys* because they work no matter what menu is active. These particular hotkeys are very handy. Other hotkeys are listed in Appendix 9. In addition to the ones we have already mentioned, you may find that a few more commonly used ones are worth memorizing, but for most things, using the submenus is clearer and easier. This is because the submenus organize the commands into groups of related commands and because they use one letter commands that suggest the meaning of the command. Many of the hotkeys give no hint of their meaning.

There are various ways to start up the editor. Provided that the main menu is active, one way is to simply press the **E** key to choose the **Edit** command from the main menu. Try it. (If the main menu is not active, first press the **F10** key and then press the **E** key.) At first it may look like nothing happened, but you will have started up the editor. If you now type something, it will appear on the screen. Try typing your name. To get rid of your name simply press the backspace key until it is erased. The backspace key is likely to be in the upper right-hand corner of your keyboard. It is likely to be marked ←, or else to have some word like “backspace” written on it.

To learn more about the editor skip over the next section, which is for version 5.0 users, and go now to the section entitled “The Editor.”

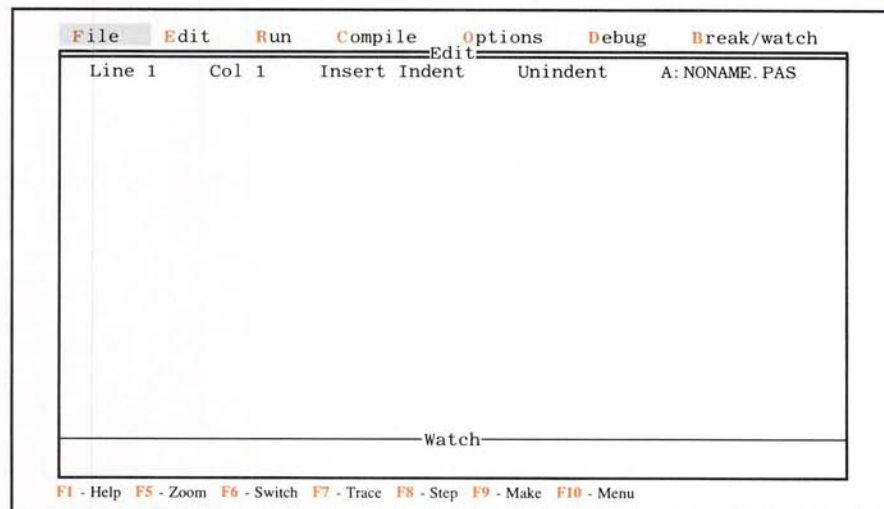
*calling the editor*

## Version 5.0

### The Menu Environment

If you have started the TURBO environment so that the copyright notice is displayed, then you are ready to start learning how to use the environment. If you press the space bar, the copyright notice will disappear and you will be left with the *main screen* display for the TURBO environment as shown in Figure A.7.3. This main screen contains

*main menu*



**Figure A.7.3**  
The version 5.0  
main screen.

two lists of commands, one on the top line of the screen and one on the bottom line of the screen. These are lists of commands available to you.

For now let us concentrate on the first line of the main screen display. This first line is called the *main menu* and contains the following entries:

**File Edit Run Compile Options Debug Break/watch**

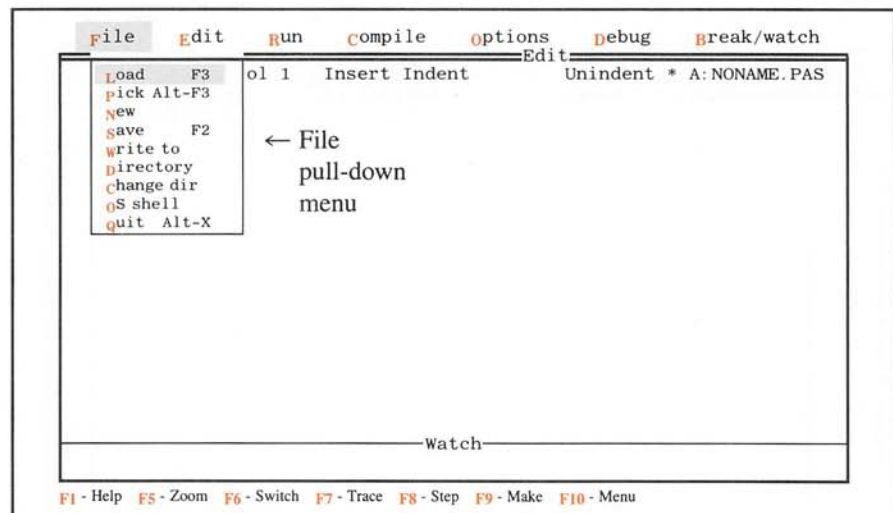
Each of these entries is the name of a command. In most situations, you can execute any one of these commands by typing the Alt version of the letter which starts the command. For example, if you press Alt-F, that will execute the File command on the main menu which causes a menu of subcommands to appear on the screen. Try it. You can skip the Edit command for the moment, but try all the rest. Try Alt-R, Alt-C, Alt-O, Alt-D, and finally Alt-B. Each of these commands produces a submenu of commands.

*pull-down  
menus*

Try pressing the Alt-F key again. That will produce the File submenu as shown in Figure A.7.4. A submenu display such as the File submenu shown in Figure A.7.4 is called a *window*. Since it seems to come down from the top of the screen it is often referred to as a *pull-down window* or *pull-down menu*. Making the window appear is called *opening* the window. When a pull-down menu is open (i.e., displayed), you can give any of the commands shown in that pull-down window. As we have already seen, there are other pull-down menus, but for now let us concentrate on the File pull-down menu. Before going on make sure that the File pull-down menu is active; in other words, make sure the screen display looks like Figure A.7.4. (Pressing Alt-F will ensure this.)

*File menu*

As long as the File pull-down menu is the active menu, the commands on that menu can be given by typing the first letter of the command. For example, if you were to press the Q key for **Quit**, that would cause you to leave the TURBO environment. This is the command you give when you are finished with the TURBO environment. (In



**Figure A.7.4**  
**Version 5.0: the**  
**File menu.**



this appendix we will use uppercase letters for these one letter commands in order to make them stand out from the rest of the text. You may use either uppercase or lowercase letters.) As a sample command, try pressing the D key to choose the **D**irectory command. This will cause yet another window to appear on the screen. It will look something like

Enter File Name

\*.\*

The **D**irectory command will list the names of the files in your current directory, but before it lists them it displays this window as a way of asking you whether you want to see all the names or just some of the names. If you simply press the return key, you will see the names of all the files in your current directory. Try it. The small window will be replaced with a larger window which contains the names of all the files in your directory.

Next try typing one of the other commands on the **F**ile pull-down menu such as **L** for Load or **N** for New. Nothing will happen. Those one letter commands only work when the pull-down menu on which they appear is active. Among all the windows on the screen, the active window is the one that was opened most recently. At this point, the active window contains no commands, only file names. In order to execute commands on a menu you must make that menu the active window. To execute a command on the **F**ile menu, you can activate the menu by pressing **Alt-F**. If you had wanted, for example, the **R**un pull-down menu instead of the **F**ile menu, then you would press **Alt-R**.

Repeat the exercise of opening the **F**ile menu and then pressing the **D** key to choose the **D**irectory command on that pull-down menu, and finally press the return key to see the list of files in your directory. As we have already seen, you cannot give any of the one letter commands on the **F**ile pull-down menu. That is because the **F**ile menu is not the active menu. To give those commands you must activate the **F**ile menu. There are short cut ways to give some of the commands on the **F**ile pull-down menu and on other menus. These short cut methods are known as *hotkeys* and they work whether or not the desired submenu is active. Look again at the **F**ile pulldown menu. After the command Load you will find **F3**. If you press the **F3** key that is the same as activating the **F**ile menu and then pressing the **L** key to choose the Load command from that menu. There is no need to memorize these hotkeys. You can always execute commands by opening menus and then choosing a one letter command. However, you may find that a few of these hotkeys are handy.

It may seem that all one needs is hotkeys and that there is no need for all these submenus (and eventually even sub-submenus). In a sense that is true. It is true provided you want to memorize a long list of commands whose names do not even hint of their meaning. (Does **F3** remind you of loading files? Does **Alt-F3** remind you of picking files?) Hotkeys are efficient and those that you would use frequently may be worth memorizing, but for most things, using the submenus is clearer and easier. This is because the submenus organize the commands into groups of related commands and because they use one letter commands that suggest the meaning of the commands.

The commands such as **Alt-F** to activate the **File** menu are hotkeys. In these cases it makes sense to use hotkeys because you will be constantly calling up these submenus. Moreover, in these cases, the hotkeys are easy to remember, since they are the **Alt** versions of the first letters in the menu names. However, if you do not want to use these hotkeys, then you can use the main menu just as you use the submenus. If the main menu is active, then you can give any of the commands on that menu by typing the first letter of the command: **F** for **File**, **E** for **Edit**, and so forth. When you start the **TURBO** environment, the main menu is active and so these commands are available to you. Returning to the main menu when it is not active can be a little tricky and so it is usually easier to use hotkeys to go directly to one of the pull-down menus. In the next paragraph we explain how to activate the main menu when it is not the active window. If the process seems confusing, you could skip that paragraph on your first reading of this appendix.

*returning to  
the main menu*

You can always make the main menu active by pressing the **F10** key. If you wish to use one letter commands on the main menu and it is not active, then simply press the **F10** key. The **F10** key switches you back and forth between the main menu and whatever else you are doing such as using the editor. Thus if you press the **F10** key when the main menu is active or if you mistakenly press it twice instead of once, you are likely to find that you are in the editor or some place else other than at the main menu. If you try to give a main menu command and it appears on the screen in the editor window, then press the **F10** key once (not twice) and try the command again.

*calling the  
editor*

We will return to explain these windows and menus in more detail, but first we will use the editor to write a Pascal program. There are various ways to start up the editor. One way is to simply press **Alt-E** to choose the **Edit** command from the main menu. Try it. At first it may look like nothing happened, but you will have started up the editor. If you now type something, it will appear on the screen. Try typing your name. To get rid of your name simply press the backspace key until it is erased. The backspace key is likely to be in the upper right-hand corner of your keyboard. It is likely to be marked **←**, or else to have some word like “backspace” written on it.

---

## The Editor

If you have been following this introduction and doing as we suggested, you are now in the editor. If you are not in the editor, enter the editor before going on. Try typing in the following Pascal program. Do not worry about why it is written as it is and do not worry about typing mistakes. We will see how to fix typing mistakes shortly. For now just try typing in the program.

```
program first;  
begin  
  writeln('hello')  
end.
```

*zoom and  
unzoom*

Now that you have typed in something using the editor you can see the purpose of the edit window. The bulk of your screen is divided into two *windows*. The top window is the editor window that contains whatever you type in. (The bottom window will be of

---



little concern to us until later.) When using the editor it is often useful to have a larger edit window. If you press the F5 key that will *zoom* the edit window so that it fills the entire screen crowding out the window below it. To get back the missing window press the F5 key again.

While in the editor, any upper- or lowercase letter you type will be entered into the file and will appear on the screen. Hence, in order to issue commands to the computer, you must use symbols other than the ordinary letters. This is one of the reasons that control characters, Alt characters, and function keys were added to the keyboard. Most TURBO editor commands consist of either one or two control characters.

As you typed in your program, you may have noticed a bit of light shaped like a square or a line which is always at the end of the character just typed. That bit of light is called a *cursor*. It is like your pen. When you type on the keyboard, what you type appears at the location of the cursor. When you give commands to delete text or otherwise change the text in the editor, the action often takes place at the location of the cursor. The cursor serves as pen, eraser, and position marker.

Now let us return to your Pascal program. If there are some typing mistakes, you must correct them before running the Pascal program. To correct a mistake, move the cursor to the location of the mistake. The cursor can move up, down, left, and right. If you have a set of four keys with arrows pointing in the four directions, you can use them to move the cursor. (These four keys should be more or less together and look like a group. There may be other keys with arrows on them which do different things.) If you do not have such arrow keys on your machine (or if you prefer to use something different), you can use the following four control characters for the four directions:

Ctrl-E  
Ctrl-S    Ctrl-D  
Ctrl-X

Notice that their location on the keyboard is an indication of the direction they move the cursor. Ctrl-E for up, Ctrl-S for left, Ctrl-D for right, and Ctrl-X for down.

To correct a mistake, you can simply delete what is wrong and type in what should be in that location. To delete a character you move the cursor to that character and then press the delete key which is a key marked something like "Del" or "rubout." If you cannot find that key, you can use Ctrl-G.

After deleting a mistake, it is easy to insert the correct text. Simply move the cursor to where you want the new text to go and type it in. The old text will get out of the way to give it room. This is called the *insert* mode. There is another mode called *overwrite* mode in which whatever you type replaces what is underneath it. To change from insert mode to overwrite mode, or vice versa, press the key marked "Ins." If you do not have an Ins key, or if you do not wish to use it, you can use Ctrl-V instead of the Ins key.

Now check your Pascal program to make sure it matches the one displayed previously. Be sure that there is a semicolon at the end of the first line and a period at the end. Also, be sure that the two quotes are the same as what is shown. They should be single quotes, not double quotes. Moreover, both quotes are the same. They are both "right quotes." Once you are sure it matches the sample we gave, you can leave the editor. To leave the editor simply press the F10 key. (An alternative way to leave the editor is to press Ctrl-K followed by Ctrl-D.)

*editor  
commands*

*cursor*

*correcting  
mistakes*

*insert  
and  
overwrite*

*leaving  
the editor*



## Version 4.0

### Running Your First Program

#### *compiler*

When you finish with the editor and wish to compile your program, you should press the C key twice (once to open the **Compile** pull-down menu and once to choose the **Compile** command from that menu.) If the compiler finds a mistake, it will give you an error message and leave you back in the editor. Correct the mistake using the editor commands as we explained above, leave the editor, and again press the C key twice. Repeat this until your program compiles successfully.

When you finally get the program to compile, the word "Success" will appear on the screen followed by the instruction "press any key." Follow the instruction and press any key. The **Compile** pull-down menu will still be on the screen. To get rid of it, press the escape key.

#### *run*

Once you have closed the **Compile** pull-down menu by pressing the escape key, you can execute any of the commands on the main menu. In particular, you can **Run** your program by pressing the R key. The program will write the word "hello" to the screen. To get back to the main menu, press any key.

#### *saving the work file*

The program which you just typed in will not be on your disk the next time that you use the system unless you *save* the file. The **Save** command is on the **File** menu. Hence, to save the file you must first activate the **File** menu. To activate the **File** menu, first make sure the main menu is active (pressing F10 will ensure this), then press the F key to choose **File** from the main menu. Once the **File** pull-down menu is displayed, you can save the file by pressing the S key to choose the **Save** command from that menu. Since the file has no name, the system will display a window saying "NONAME.PAS" or something similar. Type in any name that ends with .pas and press the return key, for example:

```
myfirst.pas
```

Your program will then be on your disk in the file named `myfirst.pas`.

If there already is a file named `myfirst.pas`, the system will ask the question

```
Overwrite MYFIRST.PAS? (Y/N)
```

If you respond with Y for "Yes," then the old contents of the file MYFIRST.PAS will be lost and your new program will be saved in that file. If you respond with N for "No," then nothing will happen, and you can repeat the process of typing S for **Save** and this time choose a different name.

#### *leaving TURBO*

To leave the **TURBO** environment, first make sure that **File** pull-down menu is displayed. If it is not, return to the main menu by pressing the F10 key and then display the **File** menu by pressing the F key. With the **File** pull-down menu displayed, you can leave the **TURBO** environment by simply pressing the Q key. In this case, Q stands for **Quit**.

Congratulations! You have just written your first **TURBO** Pascal program. You now know enough about the **TURBO** environment to start reading Chapter 2. That will teach you how to write **TURBO** Pascal programs. However, at some point you should learn

a bit more about the TURBO environment. Appendix 10 contains more detailed information about the editor. The last section of this appendix, as well as the following two appendixes, contain a bit more detail on how to manipulate files in the TURBO environment.

## Version 5.0

### Running Your First Program

When you finish with the editor and wish to compile your program, you should activate the **Compile** pull-down menu by pressing **Alt-C** and then choose the **Compile** command from that menu by pressing the **C** key. If the compiler finds a mistake, it will give you an error message and leave you back in the editor. Correct the mistake using the editor commands as we explained above, leave the editor, and compile your program again. Repeat this process until your program compiles successfully. (You do not have to leave the editor in order to use the **Alt-C** command. That command will automatically exit the editor before activating the **Compile** pull-down menu.)

*compiler*

When you finally get the program to compile, the word "Success" will appear on the screen followed by the instruction "press any key." Follow the instruction and press any key. That will leave you in the editor. You can leave the editor by pressing the **F10** key.

One way to run your program is to activate the **Run** pull-down menu and then choose the **Run** command on that menu. To do this press **Alt-R** to activate the **Run** pull-down menu and then press the **R** key again to choose the **Run** command from that menu. (You can do this while you are still in the editor. The system will automatically exit the editor before activating the **Run** menu.)

*run*

The program will write the word "hello" to the screen, but unless you have extremely quick reflexes you will not see this output. The output is flashed on the screen and then the system immediately returns you to the main screen. The output is still there however and can be seen. To see the output activate the **Run** pull-down menu and then press the **U** key to choose **User** screen from that menu. The screen may have previous input and output on in, but the last line will contain the word "hello" which is the output of your program. If you run the program again and look at the output again, there will be two lines with "hello" written on them. To return to the main screen press the **F10** key. (Actually, pressing any key will do the trick, but **F10** is a natural choice.) That will probably leave you in the editor. If you want to leave the editor, you can press the **F10** key again or you can activate one of the pull-down menus by typing the **Alt** version of the first letter in the menu name; for example, **Alt-F** will activate the **File** pull-down menu.

*finding  
your output*

The program which you just typed in will not be on your disk the next time you use the system unless you *save* the file. The **Save** command is on the **File** menu. To activate the **File** menu, press **Alt-F**. Once the **File** pull-down menu is displayed, you can save the file by pressing the **S** key to choose the **Save** command from that menu. Since the file has no name, the system will display a window saying

*saving the  
work file*



“NONAME.PAS” or something similar. Type in any name that ends with .pas and press the return key, for example:

`myfirst.pas`

Your program will then be on your disk in the file named `myfirst.pas`.

If there already is a file named `myfirst.pas`, the system will ask the question

Overwrite MYFIRST.PAS? (Y/N)

If you respond with Y for “Yes,” then the old contents of the file MYFIRST.PAS will be lost and your new program will be saved in that file. If you respond with N for “No,” then nothing will happen, and you can repeat the process of typing S for Save and this time choose a different name.

*leaving  
TURBO*

To leave the TURBO environment, first make sure the File pull-down menu is displayed. If it is not, then press Alt-F. With the File pull-down menu displayed, you can leave the TURBO environment by simply pressing the Q key. In this case, Q stands for Quit.

Congratulations! You have just written your first TURBO Pascal program. You now know enough about the TURBO environment to start reading Chapter 2. That will teach you how to write TURBO Pascal programs. However, at some point you should learn a bit more about the TURBO environment. Appendix 10 contains more detailed information about the editor. The rest of this appendix, as well as the following two appendices, contains a bit more detail on how to manipulate files in the TURBO environment.

---

## Highlighted Commands

In the TURBO environment there are a number of ways to give any one command. We have discussed using letter keys to choose commands from the active menu. It is also possible to use arrow keys to choose commands. If you study the menus you will see that one command is always highlighted. If you press the return key, then the highlighted command on the active menu is executed. For example, suppose that the main menu is active and the entry File is highlighted on that menu. Then pressing the return key will open the File pull-down menu. But suppose you want to see the Compile pull-down menu instead of the File pull-down menu. You can, of course, press the C key or press Alt-C. Alternatively, you can highlight the Compile command on the main menu and then press the return key. In order to move the highlight use the same arrow keys that you use to move the cursor when you are in the editor. Commands in the pull-down menu have a highlight that is manipulated in the same way as the highlight in the main menu. Once a pull-down menu is opened, you can move the highlight within that menu using the arrow keys. To execute the highlighted command in a pull-down menu you just press the return key.

---



## Appendix 8

# The TURBO File Menu

Whenever you are in the TURBO environment there is always one special file called the *work file*. This is the file currently available to work on. If you enter the editor, the file displayed is the work file. If you change that file, then the work file is changed. When the Compile pull-down menu is active and you choose the Compile command from that menu, it is the work file that is compiled. The work file is a kind of temporary file. It will not be on the disk the next time you use the TURBO environment unless you *save* it. Saving the work file and most other file manipulations are performed with the File pull-down menu.

*work file*

There are several ways to activate the File menu. The easiest way is to press Alt-F. Alternatively, if the main menu is active, you can choose the File command on that menu either by pressing the F key or by moving the highlight to the File command and then pressing return. (If the main menu is not active, you can activate it by pressing the F10 key.) When the File menu is activated, you will see the list of available commands for manipulating the work file.

*activating  
the File menu*

To save the work file choose the Save command. To obtain a new, blank work file choose the New command. To replace the current work file with some other file on your disk choose the Load command. (Retrieving a file in this way is called *loading* the file.) After you choose the Load command, a new window will be displayed. You should then type the name of the file you want loaded as the new work file.

*Save  
New  
Load*

If you execute the Load command, the work file will be made identical to the file you specify. For example, let us say you loaded the file TEST. PAS, then the work file will be identical to the file TEST. PAS. However, the work file and the file TEST. PAS are two different files. Any changes that you make to the work file will not be made to the file TEST. PAS. You can think of these changes as being tentative changes. If you want to make the changes permanent, you must save the work file by choosing the Save command. If you save the work file, that makes the file TEST. PAS identical to the work file and so makes your tentative changes permanent.

The File menu has a short cut method for returning to a file which you have recently used. If you choose the Pick command on the File menu, then a window will appear with a list of names showing the files which have recently been in the editor. If you use the arrow keys to highlight one of these names and then press the return key, that will load the highlighted file.

*Pick*

*Write to*

You can make any file identical to the work file. To do that, make sure that the **File** pull-down menu is displayed and then choose the **Write to** command from that menu. The system will ask you for a file name. Whatever file you name will be made identical to the work file. If you give a new name, then a new file will be created with that name. If you give the name of an existing file, then that file will be changed so that it matches the work file. However, before the system changes an existing file to make it equal to the work file, it will first ask you if you are certain that you want to overwrite that file and so lose the old contents. For example, if there already is a file called MYFIRST.PAS and you attempt to write to this file with the **Write to** command, then the system will ask

Overwrite MYFIRST.PAS? (Y/N)

If you type Y for “Yes” then the old contents of MYFIRST.PAS will be lost and MYFIRST.PAS will become identical to the work file. If you type N for “No,” then nothing will happen.

*file names*

In TURBO Pascal, file names have two parts separated by a period. (Periods are pronounced “dot” in computer jargon.) If the file contains a Pascal program, the second part of the name should be PAS. The first part of the file name is in some sense the real name. The second part is called an *extension* and is usually some sort of explanation of what kind of file it is. PAS means a Pascal program. When you change a file, the old version of the file is kept in a file with the same first name part and with BAK after the dot. BAK stands for “back-up.”

*automatic  
.PAS*

There are some technical points to keep in mind when naming files. The first part of a file name can be at most eight characters long. The second part of the name is not optional. If you type only the first name part, omitting both the dot and the second part of the name, then the system will automatically add the extension .PAS. When naming files there is no difference between uppercase and lowercase letters, so you may as well use all lowercase letters.

*Directory  
command*

If you want to see the names of your files, you can open the **File** pull-down menu and choose the **Directory** command on that menu. You will then see a window that contains two asterisks separated by a period. The TURBO system is asking you to write the name of a mask in the window. A *mask* is a description of the kind of file names you are looking for. When we used this **Directory** command in Appendix 7, we did not write anything. We simply pressed return. In that case the system uses the mask already provided, namely \*.\*, which, as we will see, describes all possible file names.

*wildcard*

The asterisk symbol \* is known as a *wildcard*. It matches any string of symbols. For example, the mask \*.PAS matches any file name that ends in .PAS, the mask \*.BAK matches any file name that ends in .BAK, and the mask MYFIRST.\* matches any file name that begins with MYFIRST.. The mask \*.\* matches any file that has anything before the dot and ends with anything after the dot. In other words, \*.\* matches any file name at all and so this is the mask to use when you want to see all file names. Since \* matches any string, it should be possible to use \* in place of \*.\* , and in fact, that is true. The simple \* does match all files. However, it is a good practice to include the dot in all masks. There are two reasons for this: You must include the dot when working in DOS and so you may as well be consistent and always include it.



Second, we usually think of the name parts as different items and so it is clearer to give a mask for each part.

The wildcard may be used anyplace in a mask. For example, the mask `S*.PAS` matches any name that begins with an S and ends with `.PAS`. Thus, among other names, it would match `SAM.PAS` and `SALLY.PAS`.

Recall that upper- and lowercase letters are not distinguished when you give file names. For example, it makes no difference whether you use the mask `*.PAS` or `*.pas`.

In version 5.0, the list of files produced by the `Directory` command can be used to pick a file to load. Simply move the highlight to the desired file and press return. The highlighted file will be loaded as the workfile. This feature is not available in version 4.0.

The `Quit` command leaves the TURBO environment. If you attempt to quit without saving the work file, the system will ask you if you wish to save the work file or not. Answer by pressing the Y key for "Yes" or the N key for "No." If you have not yet named the work file, you will be asked to name it.

The rest of this appendix describes the `Change dir` and `OS shell` commands on the `File` menu. You may wish to postpone reading about them until you feel a need for them. The first one deals with changing directories. The second one deals with moving back and forth between the TURBO environment and the DOS operating system.

Unless you do something to change the situation, all the things we did with TURBO Pascal files in Appendix 7, and thus far in this appendix, will take place on your default disk drive. If you have more than one disk drive and you wish to work on a different disk drive, then you can use the `Change dir` command on the `File` menu to change disk drives. Suppose you wish to change to disk drive B. First make sure the `File` menu is the active menu, and then choose the `Change dir` command from that window. Another window will appear that looks something like the following:

New Directory

A: \

If you then type

B:

what you type will appear in the window, replacing whatever was in there before. If you press the return key, then you will have changed to disk drive B. After that, all file names will refer to files on disk drive B. To change to some other disk drive, such as disk drive C or A, do the same thing using the desired disk drive name in place of B.

You may use the `Change dir` command to change to a subdirectory of a disk drive. For example suppose you want to change to the directory `HOMEWORK` on disk drive B, then, with the `File` menu active, choose the `Change dir` command. When the window labeled "New Directory" appears, type

B: \homework

*Quit*

*Change dir*

*changing  
directories*



After that, all file names will refer to files in that directory of disk drive B. Any full path name for a directory (as described in Appendix 5) may be used in place of \homework and that will move you to the directory specified by the path name.

*DOS  
commands*

*OS shell*

Suppose you are working in the TURBO environment and you wish to give a DOS command (or a command in some other operating system if you are not using DOS). You can simply quit the TURBO environment, give your DOS command, and then restart the TURBO environment. That is a lot of work, particularly if you must save files before quitting. You can instead use the OS shell command on the File menu. (OS stands for "Operating System.") That command will take you to DOS. When you are finished with DOS, type `exit` and you will be returned to the TURBO environment. When you return to TURBO, most things in the TURBO environment will be just as you left them.

## Appendix 9

# TURBO Hotkeys

It is possible to bypass the TURBO menu system completely. The keys listed in Figure A.9.1 are called *hotkeys*. They can be used no matter what screen is active and each will cause the action described in the table. They do not offer any new actions, just a new way of issuing commands. Unless noted otherwise, the hotkeys listed in Figure A.9.1 work in both versions 4.0 and 5.0.

Key	Action
Alt-F	Makes the File menu the active menu
Alt-E	Puts you in the Editor
Alt-R	Version 4.0: Runs your program Version 5.0: Makes the Run menu the active menu
Alt-C	Makes the Compile menu the active menu
Alt-O	Makes the Options menu the active menu
Alt-X	Quits the TURBO environment. If work file has not already been saved, asks if you want to save it.
F1	Brings up a Help window
Alt-F1	Brings up the last Help screen
Ctrl-F1	Version 5.0: Provides language help while in the editor
F2	Saves the file currently in the editor
Alt-F3	Lets you pick a file to load
F5	Zooms and unzooms the active window
Alt-F5	Version 4.0: Takes you to the saved screen Version 5.0: Takes you to the user screen
F6	Switches the active window
Alt-F6	Version 5.0: Switches the contents of the active window

**Figure A.9.1**  
**Hotkeys**

F9	Equivalent to <b>M</b> ake on the <b>C</b> ompile menu.
Alt-F9	Compiles the program in the work file.
Ctrl-F9	Version 5.0: Runs the program in the work file
F10	Version 4.0: Returns you to the main menu Version 5.0: Switches back and forth between the main menu and the active window

---

**Figure A.9.1**  
**continued**



# Appendix 10

## The *TURBO* Editor

### Commands

There are several ways to enter the editor from the TURBO environment. One way is to press Alt-E.

---

#### Basic Commands

##### Simple Cursor Movements

**One character:** arrow keys or

Ctrl-E up, Ctrl-S to left, Ctrl-D to right, Ctrl-X down

**One word:** Ctrl-A to left, Ctrl-F to right

**One page:** Ctrl-R up, Ctrl-C down

**Scroll:** Ctrl-W up, Ctrl-Z down

**To ends of line:** Ctrl-Q Ctrl-S to left, Ctrl-Q Ctrl-D to right

**To ends of screen:** Ctrl-Q Ctrl-E to top, Ctrl-Q Ctrl-X to bottom

**To ends of file:** Ctrl-Q Ctrl-R to top, Ctrl-Q Ctrl-C to bottom

**To last position of cursor:** Ctrl-Q Ctrl-P

**Automatic indenting on/off:** Ctrl-Q Ctrl-I

**Insert/overwrite switch:** Ctrl-V

(On the IBM PC, the key marked "Ins" may be used instead.)

##### Making Deletions

**Delete one character:** delete key or Ctrl-G

**Delete one word:** Ctrl-T deletes one word to the right

**Delete to end of line:** Ctrl-Q Ctrl-Y

**Delete one line:** Ctrl-Y

---

## Leaving the Editor

**Quit the editor:** Ctrl-K Ctrl-D or the F10 key

---

## Blocks—Manipulating Large Pieces of Text

You can mark a piece of text as a *block* and then copy, move, or delete the entire block, and you can also move to the block quickly. Only one block may be marked at a time. When you mark a new block, the old markings are lost. All marking, moving to, copying to, and so forth is performed at or with the cursor position.

**Mark ends of block:** Ctrl-K Ctrl-B top, Ctrl-K Ctrl-K bottom

**Mark one word:** Ctrl-K Ctrl-T

The word is then treated as a block.

**Copy block:** Ctrl-K Ctrl-C

The original block is unchanged. Markers are placed around the new block.

**Move block:** Ctrl-K Ctrl-V

The original block disappears. The new block is marked.

**Delete block:** Ctrl-K Ctrl-Y

*Warning:* There is no command to restore a deleted block.

**Read a file from disk:** Ctrl-K Ctrl-R

You will be prompted for the name of the file.

**Write block to disk:** Ctrl-K Ctrl-W

You will be prompted for a name to use on the disk.

**Hide/display block switch:** Ctrl-K Ctrl-H

On some systems, the block can be visually highlighted, and then some commands work only when it is highlighted.

**Move cursor to ends of block:** Ctrl-Q Ctrl-B to top, Ctrl-Q Ctrl-K to bottom

---

## Miscellaneous

**Indent:** Ctrl-I

Indents same amount as the line above.

**Insert one line:** Ctrl-N

Almost the same as pressing the return key.

**Restore line:** Ctrl-Q Ctrl-L

Restores the current line to its state before it was changed. Does not work once the cursor leaves the line. Does not restore an entirely deleted line.

---

**Find:** Ctrl-Q Ctrl-F

Moves the cursor to a specified string (up to 30 characters). You will be prompted for the string and any options you want. You can simply press the return key when asked for options or else choose one or more options from the following list. If you specify no option, it searches forward from the cursor position to the end of the file until it finds the first occurrence of the pattern. If the pattern is not found, nothing happens, and a message is issued.

B searches backwards from the cursor, G searches the entire text (global) independent of the cursor position, *n* (i.e., a number) finds the *n*th occurrence; U ignores the upper/lowercase distinction, and W searches for whole words only and not patterns embedded in words.

To repeat the search, type Ctrl-L.

**Find and replace:** Ctrl-Q Ctrl-A

You will be prompted for a string to search for and one to replace it with, and you will also be prompted for options. You will be asked to approve the replacement before it is made. The options are

B searches backwards, G searches the entire text (global), *n* (i.e., a number) replaces the *n*th occurrence of the search string, N replaces without asking, U ignores the upper/lowercase distinction, and W searches for whole words only and not patterns embedded in words.

To repeat the search, type Ctrl-L.

**Abort operation:** Ctrl-U

Will terminate any command, such as a find and replace, but only when it pauses for input from you.

**Insert control character:** Ctrl-P

To insert a control character into some text, type Ctrl-P followed by the character. For example, to insert a Ctrl-G, type Ctrl-P followed by Ctrl-G.

---



## Appendix 11

# Compiling to Disk in TURBO

You can compile a file for either temporary or permanent use. Compiling it for temporary use is called *compiling to memory*. If you compile to memory, then the translated compiled code for your program will not be saved on disk. If you want to use the program on another day, then you will have to recompile it. You can save the machine language compiled code on disk so that, if you return to the TURBO environment on another day, then you can run the program without having to recompile it. Compiling so that the translated code is saved on your disk is called *compiling to disk*. Both compiling to memory and compiling to disk are performed in exactly the same way: You open the C pull-down menu and then press C again in order to choose the Compile command from the C pull-down menu. Whether this results in compiling to memory or compiling to disk depends on how the Destination option is set.

If you open the C pull-down menu as shown in Figure A.11.1, you will see one line that reads

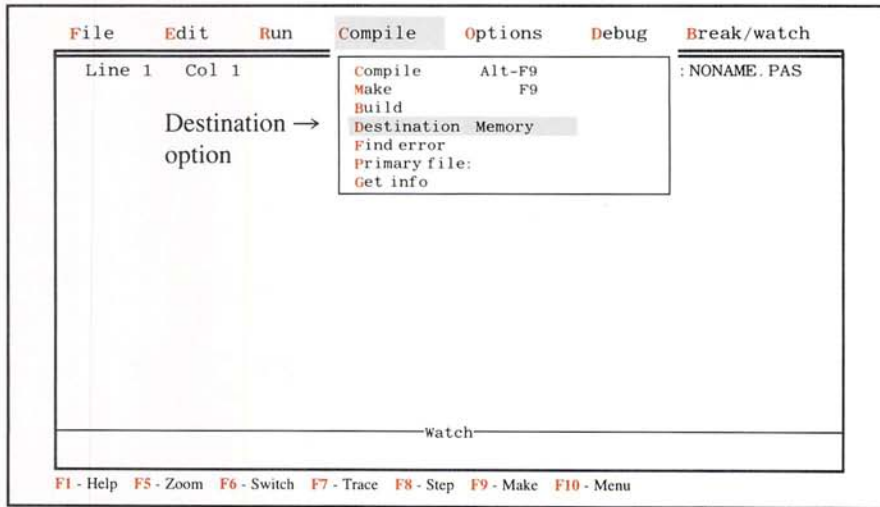
Destination: Memory

If you press the D key, the word Memory will change to Disk. If you press the D key again, the word Disk will change back to Memory. The D key switches between the words Memory and Disk. If the word displayed is Memory, then when you compile a program it will compile to memory. If the word displayed is Disk, then when you compile a program it will compile to disk.

When you compile to disk, the translated machine language version of your program is stored in a file whose name ends in .EXE, which stands for "executable." For example, if your Pascal program were in a file called MYFIRST.PAS and you compile it to disk, then the machine language version will be in a file called MYFIRST.EXE. The file with the machine language version inherits its first name part from the file with the Pascal program, but it ends in .EXE rather than .PAS.

When a program is compiled to disk, it can be run from DOS without even entering the TURBO environment. For example, suppose you compile the program in MYFIRST.PAS to disk so that it is in the file MYFIRST.EXE. If you then quit the TURBO environment by typing Q for "Quit" while the F menu is displayed, you will get a DOS prompt that looks something like

---



**Figure A.11.1**  
The Destination  
option on the  
Compile menu.

A>

You can run your program from DOS by simply typing `myfirst` after the prompt like so

A>`myfirst`

When you enter this command you are running the (translated) program in `MYFIRST.EXE`, and so that file must be on the disk in disk drive A. However, the file `MYFIRST.PAS` need not be present. Be sure to note that, although you are running the program in `MYFIRST.EXE`, you do not type the ending `.EXE`.

You may move the file with the translated machine code, `MYFIRST.EXE` in our example, to any disk or directory and run it from there. For example, if you move the file `MYFIRST.EXE` to disk drive B, and change the default drive to B, then you can do the same thing on drive B. After moving the file to drive B, the following would run the program:

B>`myfirst`

The file `MYFIRST.PAS` no longer has anything to do with this. It may still be on drive A or it may even have been erased or moved to a backup disk.

It may seem pointless to run a program by quitting the TURBO environment and then typing the program name. In fact, this example was just an exercise. However, there are times when you might want to run a program from DOS. You might want to develop a program that is used by somebody who does not know the TURBO environment, or you might have a program which you use frequently and for which you do not want to bother with the TURBO environment.

## Appendix 12

# Input/Output Redirection in DOS

In most DOS operating systems, you can run a TURBO Pascal program so that it takes input from a specified file and writes its output to another specified file. The program is written in the normal manner, just as if it took input from the keyboard and sent output to the screen. However, if the same program is run in a different way, then it will instead take input from a file and give output to some other file.

For example, consider the program in Figure A.12.1. It can be run, and if the user types two integers at the keyboard, the program will write their sum to the screen. Now suppose that the file DATA.TXT contains two numbers and you wish to run this program so that it adds these two numbers and writes their sum to the file RESULT.TXT. Assuming that the files are on the default disk drive, you can do this as follows:

1. Compile the program to disk as described in Appendix 11. For example, let us assume that the program is compiled to the file ADD.EXE.
2. Return to the DOS system by quitting the TURBO environment so that you see a DOS prompt such as

A>

3. Execute the following DOS command:

ADD < DATA.TXT > RESULT.TXT

If the file DATA.TXT contains the numbers 2 and 3, then after the above DOS command is executed, the file RESULT.TXT will contain the number 5. This process will

```
program AddUp;
var N1, N2, Sum: integer;
begin{Program}
  read(N1, N2);
  Sum := N1 + N2;
  write(Sum)
end. {Program}
```

**Figure A.12.1**  
**Sample program.**



create the file RESULT. TXT if it did not already exist. If it did already exist, then its old contents are lost. In general, the form of this redirection command is

```
program < input file > output file
```

This command must be given in the DOS environment. The “program” must be the name of the file containing the . EXE version of the program.

You may follow these instructions without understanding what they mean, but to dispell some of the mystery, we will give a brief description of what is involved in this process.

Redirection is a general DOS facility that can be used with any program in a . EXE file. It simply says to run the program taking input from some specified file and sending the output to some other specified file. The syntax for the command is natural, provided you think of the symbols “<” and “>” as arrowheads. It is easiest to understand the input and output parts separately, and so we will discuss them separately.

If you wish to run a program with input from the keyboard and output to a file, you can do so. For example, if the program ADD. PAS is compiled to disk as in the above example, then the following DOS command will run the program, taking input from the keyboard and sending output to the file RESULT. TXT:

```
ADD > RESULT. TXT
```

If you type the numbers 3 and 12 at the keyboard, the program will follow the arrowhead and send its output to the file RESULT. TXT, and that file will contain the number 15 after the program is run.

If you wish to run a program that takes input from a file and sends its output to the screen, you can do so. For example, if the program ADD. PAS is compiled to a . EXE file as in the above examples, then the following DOS command will run the program, taking input from the file DATA. TXT and sending output to the screen:

```
ADD < DATA. TXT
```

In this case think of the arrowhead as pointing in the direction of the data flow from the file to the program. If the file DATA. TXT contains the numbers 2 and 3, then the program will output the number 5 to the screen.

If you now want the program ADD to take its input from the file DATA. TXT and to send its output to the file RESULT. TXT, you simply use both arrows to produce

```
ADD < DATA. TXT > RESULT. TXT
```

One final warning is in order. Although Input/Output redirection will work for almost any program that you might reasonably want to use it on, there are programs for which it will not work. In particular, you cannot use Input/Output redirection if your program uses the predefined unit crt.

*redirecting output*

*redirecting input*

*warning  
program*

## Appendix 13

# The *TURBO* Crt Unit

TURBO Pascal has a predefined unit call `crt` which is available on IBM PCs, XTs, ATs, PS/2s, and true compatibles. The following procedures are in `crt` and are explained in detail elsewhere in this book: `clrscr`, which clears the screen, is explained in Chapter 10, `gotoxy`, which moves the cursor, is explained in Appendix 14, and `window`, which opens a text window, is explained in Appendix 15. Below we describe some of the other items in this unit. Using `crt` will also speed up screen output. However, you cannot use DOS Input/Output redirection (Appendix 12) with programs that use this unit. The term "CRT" is a common abbreviation for "Cathode Ray Tube," which refers to the terminal screen. The unit is called `crt` because the various predefined procedures and functions have to do with terminal input and output. In order to use any of the items in `crt` your program must include a *uses* clause such as the following immediately after the program heading:

```
uses crt;
```

### Ctrl-Z as end-of-file

Syntax:

```
checkeof
```

Example:

```
checkeof := true
```

`checkeof` is a special boolean variable that is used by the *TURBO* input routine to control how the boolean `eof` responds to Ctrl-Z. The variable `checkeof` should not be declared, but it can be used just like any boolean variable. If you do nothing to set or change the value of `checkeof`, then it will have the value `false`. In this default setting, there is no way to make `eof` true by typing something from the keyboard. If you change the value of `checkeof` to `true`, then Ctrl-Z will act as an end-of-file marker when typed in at the keyboard. If `checkeof` is `true` and the user types Ctrl-Z, then `eof` will be set to `true` at the next `read` or `readln`.

### timed pause

Syntax:

```
delay (<milliseconds>)
```

---

Example:

```
delay (500)
```

Causes the program to pause for approximately the number of milliseconds given as the argument. <millisecond> is an expression of type integer. A complete example is given in Figure A.13.1.

### sound

Syntax:

```
sound (<hertz>)
```

Example:

```
sound (440)
```

Sounds a tone of the frequency given in hertz by its argument. <hertz> must be an integer expression. The sound continues until it is shut off by the procedure nosound. A complete example is given in Figure A.13.1.

```
program Roadrunner;  
{Demonstrates the procedures delay, clrscr, and sound.}  
uses crt;  
const Frequency = 600; {hertz}  
      Minute = 1000; {milliseconds}  
      FourthMinute = 250;  
      EighthMinute = 125;  
  
procedure Beep;  
begin{Beep}  
  sound(Frequency);  
  delay(FourthMinute);  
  nosound  
end; {Beep}  
  
begin{Program}  
  writeln('Here comes the roadrunner!');  
  delay(Minute);  
  clrscr;  
  
  Beep;  
  delay(EighthMinute);  
  Beep;  
  
  writeln('Bye')  
end. {Program}
```

**Figure A.13.1**  
TURBO Pascal  
program with  
sound.



**stop sound**

Syntax:

`nosound`

Stops the tone started by `sound`.

**locating the cursor**

Syntax:

`wherex  
wherey`

Example:

`X := wherex;  
Y := wherey`

Two functions of zero arguments that return the *x* and *y* coordinates of the current cursor position. For example, with *X* and *Y* set as in the example, the program can, at a later time, return to this cursor location by executing `gotoxy (X, Y)`.

**clear to end of line**

Syntax:

`clreol`

Clears the current screen line from the current cursor position to the end of the line. Typically used in conjunction with `gotoxy`.

**insert line**

Syntax:

`insline`

Inserts an empty line at the cursor position.

**delete line**

Syntax:

`delline`

Deletes the line that currently contains the cursor.

**test for “press any key”**

Syntax:

`keypressed`

A boolean-valued function with no arguments. Returns `true` if a key has been pressed at the console but the program has not yet read the keypress; otherwise it returns `false`.

---

**read a character with no screen echo**

Syntax:

`readkey`

Example:

`Ans := readkey`

This is a function of zero arguments. When the function is evaluated, the program pauses and waits for the user to type a character at the keyboard. The first character typed is returned as the value of the function `readkey`. This character is not echoed on the screen. For example, if you wanted to create a user interface like the TURBO menu system, you could use this function to read one letter commands.

---

# Appendix 14

## TURBO Pascal—

### Character Graphics

Figure A.14.1 illustrates the use of `gotoxy` to create simple graphics. This sort of graphics is called *character graphics*, since it consists of ordinary characters placed at various locations on the screen. Such graphics can be done without `gotoxy`, but is much simpler to do with the aid of `gotoxy`. With `gotoxy` your program can “back up” and add more to the drawing. Notice that the two sides of the triangle are drawn separately, much like you might draw them with a pencil. First the left side is drawn; then the cursor returns to the top of the screen and draws the right side. Be sure to note that character graphics requires the predefined unit `crt`.

*screen  
coordinates*

The upper left-hand corner of the screen has coordinates (1, 1). Coordinates are positive to the right and down. One unit in the *x* (i.e., horizontal) direction has size equal to one character width. One unit in the *y* (i.e., vertical) direction has size equal to the height of one line. Suppose, as is typical, that the screen contains 80 characters per line and 25 lines per screen. The upper left-hand corner of the entire screen has coordinates (1, 1), the upper right-hand corner has coordinates (80, 1), the lower left-hand corner has coordinates (1, 25), and the lower right-hand corner has coordinates (80, 25).

#### Program

```
program CharGraphics;
{Demonstrates TURBO character graphics, using gotoxy,
by drawing a Christmas tree.}
uses crt;

procedure Box(X1, Y1, X2, Y2: integer);
{Draws a rectangle with upper left-hand corner
at (X1, Y1) and lower right-hand corner at (X2, Y2).}
var X, Y: integer;
```

**Figure A.14.1**  
Character graphics  
program.



```

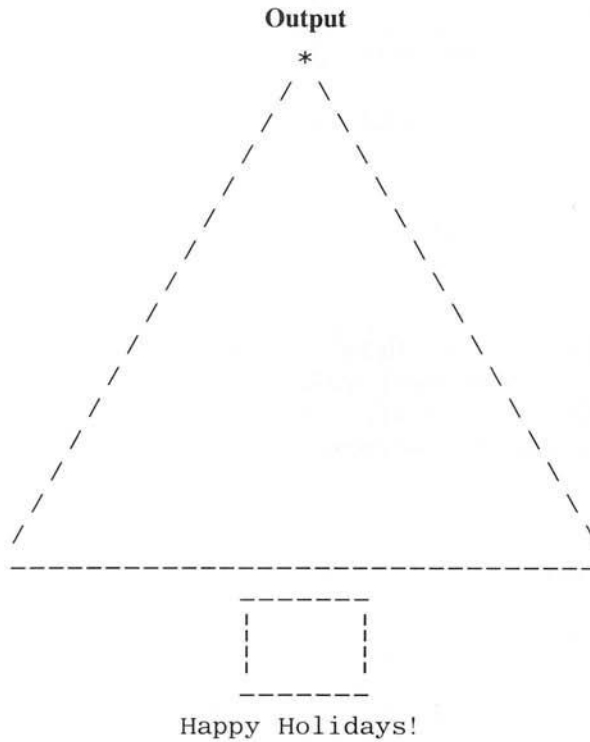
begin{Box}
  {Draw top}
  gotoxy(X1, Y1);
  for X := X1 to X2 do
    write('-');
  {Draw sides}
  for Y := Y1+1 to Y2 - 1 do
    begin
      gotoxy(X1, Y);
      write('|');
      gotoxy(X2, Y);
      write('|');
    end;
  {Draw bottom}
  gotoxy(X1, Y2);
  for X := X1 to X2 do
    write('-');
end;{Box}

procedure Triangle(TopX, TopY, Height: integer);
{Draws a triangle with top point at (TopX, TopY), height of
Height, and base of length 2*Height. Since x and y scales are
not the same, it will look taller and thinner than expected.}
var X, Y: integer;
begin{Triangle}
  {Draw left side}
  for Y := 1 to Height do
    begin
      gotoxy(TopX - Y, TopY + Y);
      write('/');
    end;
  {Draw right side}
  for Y := 1 to Height do
    begin
      gotoxy(TopX + Y, TopY + Y);
      write('\');
    end;
  {Draw base}
  gotoxy(TopX - Height, TopY + Height);
  for X := 1 to 2*Height + 1 do
    write('-');
end; {Triangle}

```

**Figure A.14.1**  
continued

```
begin{Program}  
  clrscr;  
  gotoxy(40, 1); write('*');  
  Triangle(40, 1, 15);  
  Box(37, 17, 43, 20);  
  gotoxy(33, 22);  
  write('Happy Holidays!')  
end. {Program}
```



**Figure A.14.1**  
continued

## Appendix 15

# *TURBO Pascal—Text Windows*

On machines that support the predefined unit `crt` (IBM PC, XT, AT, PS/2, and true compatibles), your TURBO Pascal programs may define windows on the screen. A *window* is simply a portion of the screen that serves as the whole screen. It is a way to define a small screen within the screen. The advantage of this facility is that a program can have more than one window on the screen at the same time. For some applications this capability can be as powerful as having more than one screen. For example, one screen location might contain permanent instructions such as

Type A to Add to inventory, S to Show stock, Q for Quit.

A window on another part of the screen can then be used to interact with the user. Since the window is treated as a separate screen, it will never cause the permanent instructions to be scrolled off the screen. When the window is full, text will scroll up in the window just as if the window were a screen; the permanent instructions, which are outside of the window, remain untouched, no matter how much text passes through the window. The program in Figure A.15.1 illustrates this technique, but to understand it you must first understand the window command.

A call to the window command is of the form

```
window(x1, y1, x2, y2)
```

where all four arguments are value parameters of type `integer`. This command opens a rectangular window whose upper left-hand corner has coordinates (*x1*, *y1*) and whose lower right-hand corner has coordinates (*x2*, *y2*).

Suppose, as is typical, that the screen contains 80 characters per line and 25 lines. The upper left-hand corner of the entire screen has coordinates (1, 1), the upper right-hand corner has coordinates (80, 1), the lower left-hand corner has coordinates (1, 25), and the lower right-hand corner has coordinates (80, 25). To open a screen consisting of the lower right one-quarter of the screen, you would use

```
window(40, 13, 80, 25)
```

Once a window is opened, all of the screen outside the window is frozen, and the window becomes the screen. Any text that is sent to the screen will appear in the win-





dow. If you wish to have your program go back and write on part of the screen outside the window, you can have the program open a second window in a different location. Once the second window is opened, all the text written on the screen will appear in this second window.

Our description makes it sound as if you cannot send output to two windows simultaneously. Strictly speaking this is true, but you can go back and forth between windows by reopening the windows. For all practical purposes this is the same as having more than one window producing output at the same time. If you do not clear the screen in a window, then the text in that window remains intact until your program returns to the window and adds more output to the text in that window. However, when your program returns to a window, it may need a `gotoxy` to position the cursor so that writing will resume at the appropriate location in the window.

Notice that at the end of the program in Figure A.15.1, we have a call that makes the entire screen the window. This call ensures that the screen will behave correctly for the next program that is run. Leaving a small active window at the end of a program can cause garbled output when the next program is run.

*multiple windows*

*restoring  
the screen*

---

## Appendix 16

### Some Common *TURBO Pascal* Compiler Directives

A number of details concerning the manner in which the compiler translates a *TURBO Pascal* program are controlled by *compiler directives*. A compiler directive is not part of the Pascal program. However, it affects what the compiler does. In many cases this looks the same as if the directive changed the meaning of the Pascal program in some way.

There are two ways to set compiler directives. One method is to insert a special comment into the program. The other method is to use the options menu. We will discuss the comment method in this appendix. If you wish to use the menu, then from the main menu, type O for Options and then C for Compiler. With the aid of the summaries listed at the end of this appendix, the use of the menu should be more or less self-evident.

In Chapters 8 and 9 we discussed the R compiler directive and gave the syntax for inserting it into a *TURBO Pascal* program. The syntax for other directives is similar. Directive comments all start with the comment delimiter "{", followed by the \$ symbol, followed by a one letter code for the directive, sometimes followed by a plus or minus sign, *all with no spaces*. After that there may be a space followed by some parameter(s), and/or a comment, and finally the closing comment delimiter "}".

Many directives act like a switch telling the compiler to act in one of two possible ways, indicated by a plus or minus sign. For example, {\$B+} turns the B directive switch "on" and {\$B-} turns it "off." Each such switch directive has a default setting and if you do nothing to change it, the switch will be at this default setting. For most programs, the default setting will work well. The directive switches given below may be placed anywhere in a program and may be reset within the program. Some other switches (given in the *TURBO* manual) must be set once for the entire program.

---



**B: Boolean Evaluation****Default:** {\$B-}

If you use the directive {\$B+}, then all parts of all boolean expressions will be evaluated; this method is called *complete* evaluation. If you use the default setting {\$B-}, or equivalently no B directive at all, then the code generated by the compiler will evaluate only as many subexpressions as it needs in order to determine the value of the expression; this method is called *short circuit* evaluation. For example, consider the following:

(N = 1) and (X/Y > 7)

If N has a value other than 1, then with short circuit evaluation, i.e., with {\$B-}, the computer will not even consider the second part of the expression and will return false, even if the second part cannot be evaluated. With complete evaluation, i.e., with {\$B+}, the second expression will always be evaluated, and if the value of Y is 0, will produce an error.

All the programs in the text will work with either setting. Any program written to work with complete evaluation, {\$B+}, will also work with the other setting. Hence, for portability, the text programs are all written in that fashion.

**I: Input/Output Error Handling****Default:** {\$I+}

{\$I-} allows the program to be written with explicit instructions for handling I/O errors. See Appendix 17 for details.

**I: Include Files****Syntax:** {\$I <file name>}

Whenever the comment {\$I <file name>} occurs in a program it is as if the contents of the file named in the directive were inserted into the program file at that point. The named file should end in .PAS. The named file must contain complete declarations. It cannot contain incomplete procedure or other declarations. It cannot contain portions of the main body of the program. Include files may be nested to a depth of eight.

**R: Index Range Check****Default:** {\$R-}

In the default setting, {\$R-}, array indexing operations are not always checked to see if they are within defined bounds, nor are assignments to scalar and subrange types always checked to see if they are within range. An index error is thus likely to produce unpredictable results. The setting {\$R+} performs these checks but is likely to slow the program down. It is a good idea to use {\$R+} while debugging a program.

**S: Stack Checking****Default:** {\$S+}

With the default option, {\$S+}, a check is made to insure that space is available for local variables on the stack before each call to a subprogram. When turned off, {\$S-}, no checks are performed, but *if the program is correct and does not exceed machine capacities*, then it should run faster.

**V: Var String Parameter Type Checking****Default:**  $\{\$V+\}$ 

In the default setting,  $\{\$V+\}$ , actual and formal variable parameters of *string* types are checked for length. If they do not match, an error occurs. When turned off,  $\{\$V-\}$ , actual parameters whose lengths do not match their corresponding formal variable parameters are allowed.

---

# Appendix 17

## TURBO Pascal— I/O Error Handling

Input and output are usually referred to collectively as *I/O*. Errors involving I/O to or from a program, file, or device are called *I/O errors*. If the user enters input in an incorrect format, or if any other I/O error occurs, then your program will terminate, and an error message will be issued. This can happen even if the program contains no error. For example, if the user asks the program to open a nonexistent file using `reset`, then an I/O error will occur. This error could result because the user misspelled the name of the file and so accidentally entered the name of a nonexistent file. Rather than terminate the program when an I/O error occurs, it might be preferable for the program to take some action based on the particular error that occurred. In this case it would make sense to ask the user to retype the file name. TURBO Pascal provides a mechanism to do this sort of error handling.

The `I` compiler directive can be used to turn off error checking. If you insert the following anyplace in a program, then from that point on, any I/O errors will not cause the program to terminate:

*The I directive*

```
{ $I - }
```

You can turn error checking off and on any number of times within a program. The above directive can be inserted anyplace in a program, and it will turn off error checking until control reaches the following directive, which turns error checking back on:

```
{ $I + }
```

If you turn error checking off with the `{ $I - }` directive, then the program will proceed despite finding an I/O error. However, it usually is pointless for the program to proceed normally after such an error. Hence, you should provide some additional code which causes the program to branch and take some appropriate action, such as redoing the I/O, before it proceeds. To facilitate such branching TURBO Pascal provides a predefined function called `ioresult`. This is a function of zero arguments that returns the error number of the last I/O error made by the program.

*ioresult*

The I/O error numbers can be found in the TURBO Pascal manual. However, typically all one needs to know is whether or not an error has occurred, and that can be done by testing `ioresult` to see whether it is zero or nonzero. If no I/O error



has occurred, `ioresult` returns zero. (Zero is not an error number). Hence, if `ioresult` returns zero, then no error has occurred and the program can safely proceed normally. If `ioresult` returns a nonzero value, then the program should branch to handle the error. After each call (each “check”) of `ioresult`, the value of `ioresult` is reset to zero.

One common use of `ioresult` is to check for mistyped file names as in the following sample loop. The variable `Error` is of type `integer`. The other variables are of the obvious types. The variable `Error` is needed because the program needs to check for an error twice. Any one check of `ioresult` will reset it to zero. Hence, the value must be remembered in the variable `Error`. Notice that the compiler directive can include a blank and an explanatory comment after the directive. However, there should be no blanks between the initial curly bracket and the plus or minus sign.

```
repeat
  writeln('Enter file name: ');
  {$I- turns off error checking.}
  readln(FileName);
  assign(FileVar, FileName);
  reset(FileVar);
  {$I+ turns error checking back on.}
  Error := ioresult;
  if Error <> 0 then
    writeln(FileName, ' cannot be found. Try again. ');
until Error = 0;
```

# Appendix 18

## Version 5.0-The Integrated Debugger

Version 5.0 of TURBO Pascal includes a debugger that lets you step through your program line by line and lets you see the values of variables and expressions change as the program is executing. That means that you can watch your program run from the inside. You can, for example, see the value of any variable at any point in the computation, whether or not the program contains an output statement to write it to the screen.

The heart of the debugger can be understood as soon as you learn the basics of Pascal and the TURBO menu environment. The first two sections of this appendix cover most of the features of the debugger. They can be read anytime after you have read Appendix 7 on the menu environment and Chapters 1 through 3 which cover the basics of the Pascal language. The remaining sections include some extra detail on more advanced Pascal constructs. Each of those sections can be read as soon as you are introduced to the items mentioned in the section title.

---

### Introduction to the Debugger

In order to see how the debugger works, you need a program to use it on. We will take you through a sample debugging session using the program shown in Figure A.18.1. Before going on, you should type in that program using the editor and save the file. Type in Figure A.18.1 just as it is shown. If you think it contains an error, do not correct it.

The compiler can generate different kinds of machine language code from your Pascal program. If you only intend to run your program the code can be simpler than it needs to be for the debugger. The kind of code produced by the compiler is controlled by various switches. Normally, these switches are already set to the correct position for generating code suitable for the debugger. However, it is a good idea to check the switches before you use the debugger for the first time. To check the first switch activate the **Debug** pull-down menu so that screen look like Figure A.18.2. (Pressing **Alt-D** is one way to do that.) Notice the entry **Integrated debugging**. The

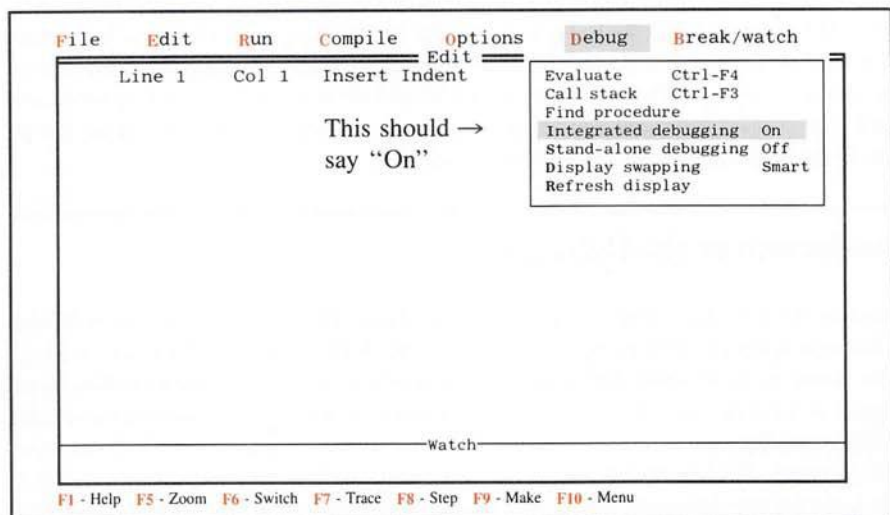
*setting  
switches*

```

program Sample;
var N1, N2, Sum: integer;
    Ave: real;
begin
  writeln('Start Sample of Debugger');
  {The next 3 lines are only here to demonstrate the debugger.
  They would not be included in a normal program.}
  N1 := 9;
  N2 := 10;
  Sum := N1 + N2;
  writeln('Enter two numbers and');
  writeln('I will compute their average. ');
  readln(N1, N2);
  Sum := N1 + N1;
  Ave := Sum/2;
  writeln('The average of ', N1, ' and ', N2, ' is ', Ave :5:2);
  writeln('End Sample of Debugger')
end.

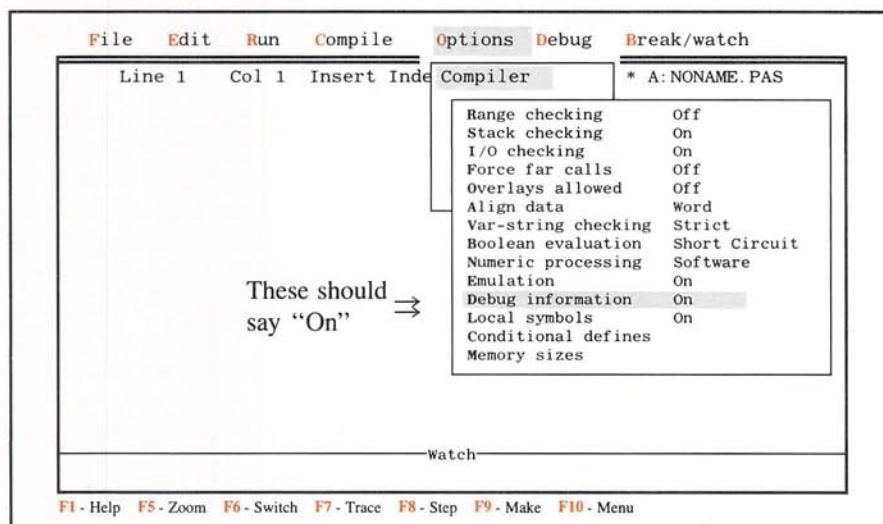
```

**Figure A.18.1**  
Program to  
demonstrate the  
debugger.



**Figure A.18.2**  
The Debug menu





**Figure A.18.3**  
The  
Options\Compiler  
menu

word On should be to its immediate right. If it is not, then press the I key. That key switches the integrated debugger on and off. Since you will be using the integrated debugger, you want it on. There are two other switches that need to be on. To find those other switches activate the Options pull-down menu. (Pressing Alt-O is one way to do that.) When the menu is displayed choose Compiler from that menu. This will produce a new sub-menu with a long list of commands as shown in Figure A.18.3. Notice the entries Debug information and Local symbols. The word On should be to the right of each of these. If it is not, then press the first letter of the command. That switches between Off and On. After you are certain that all these switches are On, you can simply forget them. You need not turn any of them Off when you finish using the debugger. If you leave them on, then the next time you use the debugger, these switches should still be on. You need not check them again, as long as you know that nobody has changed them.

That is all that we will be doing with the Debug menu and the Options menu. From now on, we will be using only the Run menu and the Break watch menu. To activate these menus press the Alt version of the first letter in the menu name, Alt-R or Alt-B. The Break watch menu is used to specify the variables and expressions that you wish to trace (i.e., watch change). The commands for actually running the debugger are on the Run menu. Even if you use hotkeys to give commands, you will still want to be aware of these pull-down menus. Most commands on these menus also list the equivalent hotkey, so the menus can be used to look up a hotkey if you forget it.

Watching the values of a variable change as the program runs is called *tracing the variable*. You can give the debugger a list of variables and it will trace them for you automatically. To get started on the process of naming the variables to be traced, first activate the Break watch pull-down menu. (Pressing Alt-B is one way to do that.) Next, choose the entry Add watch from this menu. (Pressing the A key is one way to do that.) A window will appear that looks something like the following:

Watch  
window

## Add Watch

The window on the bottom of your screen is called the Watch window. It contains the variables to be traced. This new window is called the Add Watch window. It is used to name a variable that you want to add to the Watch window. Type in N1 which is the name of a variable in your program. The variable will appear in the Add Watch window as you type it. (If the Add Watch window already contains an identifier, just ignore it for now. It will go away when you start typing.) After typing in the variable, press the return key. The variable N1 now appears at the bottom of the screen in the Watch window. You have just told the debugger that you want to trace the variable N1. At this point the variable N1 may be followed by some worrisome phrase, such as "Unknown identifier." Do not be disturbed by this. Since you have not yet compiled your program, the debugger does not know what N1 is and is telling you that it does not know about it. That will change shortly.

There are a number of short cut ways to add identifiers to the Watch window. In order to try out the first one, be sure the **Break watch** menu is active. (Press **Alt-B** if it is not.) Notice that the **Add watch** entry has the hotkey **Ctrl-F7** written after it. You can use this hotkey (instead of the pull-down menu) to add variables to the Watch window. If you forget the hotkey, you can use the pull-down menu to remind yourself of what it is. Press the hotkey **Ctrl-F7**, the result is the same as if you had pressed the **A** key. However, **Ctrl-F7** will work whether or not the **Break watch** pull-down menu is active. To see this, press **Alt-E**. That will leave you in the editor. Now type **Ctrl-F7**. You will immediately see the **Add watch** window. This time type in N2 and press return. That will add the variable N2 to the Watch window.

There is another short cut that will save you the bother of typing in variable names. You should now be in the editor. If you are not, or are not sure, press **Alt-E**. Now move the cursor so it is in the identifier **Sum**; anyplace in the identifier will do. After positioning the cursor, press **Ctrl-F7**. The **Add Watch** window will appear and will already contain the identifier **Sum**. Simply press the return key and the variable **Sum** will be added to the Watch window. You can add any variable to the Watch window in this way. Before going on, add the variable **Ave** to the Watch window in this way.

You can only add one variable at a time to the Watch window. Do not type in a list of variables. Instead, you should add them one at a time in, more or less, the manner we went through in this example.

The Watch window is not limited to variables. You can also place arithmetic expressions in the Watch window. For example, you might add the following expression to the Watch window:

N1 + N2

Simply type the expression in exactly as you would type in the name of a single variable. Press **Ctrl-F7** to produce the window labeled **Add Watch** (or use menu commands to accomplish the same thing). When that window is displayed, type in the expression and press the return key.

*expressions  
in Watch window*

*starting  
over*

If you made a mistake in adding variables to the Watch window, you can clear the window and start over again. To clear the Watch window choose **Remove all watches** from the **Break watch** menu. (Press **Alt-B** and then the **R** key.)

After a few preparatory paragraphs we will tell you how to start a debugging session. If you become confused, or if for any other reason you wish to end the session, activate the **Run** menu and choose the **Program reset** command from that menu. Alternatively, simply press the hotkey **Ctrl-F2**. Those are two ways to give the same command. The hotkey is easier once you learn it, but the menu command is easier to remember. Memorize or write down one of those two ways to end a session before going on.

If in the middle of a session, you find yourself in the editor or otherwise seem to be outside of the debugging session, press the **F7** key. If a window appears demanding an answer to the question “Source modified, rebuild? (Y/N),” press the **Y** key and the session will start over. If things are not right, terminate the session (either **Ctrl-F2** or **Run** followed by **Program reset**). After that, use the editor to fix any changes you made to the file and then start over.

You are now ready to step through this program and watch the debugger trace the variables. The basic debugging commands are summarized in Figure A.18.4. In that table, menu commands are specified by giving the pull-down menu containing the command followed by the command to choose from that menu. For example, **Run\Trace** means first choose the **Run** menu and then choose **Trace** from that menu.

Hotkey	Equivalent Menu Command Action
<b>F7</b>	<b>Run\Trace</b> Compiles program and initiates debugging session. When in a session, each press of <b>F7</b> executes the next line and updates the watch window.
<b>Alt-F5</b>	<b>Run\User screen</b> Brings up the user screen. (Toggles between screens.)
<b>Ctrl-F7</b>	<b>Break watch\Add watch</b> Adds an expression to the watch window.
No hotkey	<b>Break watch\Delete watch</b> Deletes an expression from the watch window.
No hotkey	<b>Break watch\Remove all watches</b> Deletes all expression from the watch window.
<b>Ctrl-F2</b>	<b>Run\Program reset</b> Ends a debugging session.

**Figure A.18.4**  
**Basic Debugger**  
**Commands**



*starting  
a session*

To start a debugging session press the F7 key. Your program will be compiled and when the compilation is completed, the line with word *begin* will be highlighted. Notice that each variable has a number after it. That number is the value of the variable. When the program starts these values might be anything. At the start of the program the variables are uninitialized and so they contain whatever values happen to be left in memory from the last time the computer was used. As the program is run, these values will change so that you can see how the program changes the values of variables. This is what we meant when we said the debugger would automatically trace variables for you.

*user  
screen*

You are now ready to use the debugger on your program. Every time you press the F7 key one line of your program will execute. The first time that you press it, the line with the *begin* is executed. That line does nothing, but the highlight will be moved to the first *writeln*. The highlight shows the line to be executed next. In order to execute this *writeln* simply press the F7 key one more time. The *writeln* will output a message to the screen, but the message will go by so rapidly that you will probably miss it. To see the output press Alt-F5. Alternatively, activate the Run menu and choose User screen. The screen that shows your program's input and output is called the *user screen*, since it is the screen that interacts with the user of the program. You will frequently want to look at the user screen, so the hotkey Alt-F5 is easier to use than the menu. If you forget which hotkey to use, simply activate the Run menu. It shows the hotkey right after the entry User screen. To return to the main screen press Alt-F5 again. You can toggle back and forth between the main screen and the user screen by pressing Alt-F5. At this point, the user screen may contain things left over from previous interactions, but the last thing on the screen should be the following line which was produced by that first *writeln* in your program:

Start Sample of Debugger

*stepping  
through*

Every time you press the F7 key another line is executed and the highlight moves to the next line. The line is executed as the highlight leaves the line, not as it moves onto the line. When the highlight is on a line it has not yet been executed. Try pressing the F7 key until the highlight reaches the next *writeln*. Notice that, when an assignment statement is executed, the value of the variable on the left-hand side of the assignment operator changes in the Watch window.

*adding to  
Watch window*

You can add a variable or expression to the Watch window at anytime, even in the middle of a debugging session. As an exercise, add the following expression before going on:

$(N1 + N2) / 2$

Notice that, as soon as you add it to the Watch window, it is evaluated and its value is shown.

Next, press the F7 key until the highlight is on the *readln*. If you press the F7 key one more time, the *readln* statement will be executed and the screen will switch to the user screen. The computer is now waiting for you to type in two numbers so that the program can proceed. Type in the two integers 2 and 4, separated with a blank, and then press the return key. The main screen will return. Note that the variables N1 and N2 in the Watch window now have values equal to the two numbers you typed in.

---

Continue to press the F7 key until the highlight reaches the next to last `writeln`. If you press the F7 key one more time and then press Alt-F5 the user screen will show the programs output, which proclaims that the average of 2 and 4 is 2.00. Something is wrong.

The error in this program is not hard to find even without the debugger, but let us pretend that you do not know what is wrong. If you look in the Watch window you will see that value of Sum is 4, but the value of `N1 + N2` is 6. The value of Sum is incorrect. If you look at the last statement that reset the value of Sum, you will see that, instead of adding N1 and N2, the statement adds N1 to itself. Even obvious bugs like this one are sometimes missed. Other bugs are much more subtle. The debugger can help you locate a wide range of different kinds of bugs.

Once you have located a bug (or at any other time), you can terminate the debugger session by pressing Ctrl-F2 (or alternatively choosing **Program reset** from the **Run** menu.) Of course, the one bug that you found need not be the only bug in a program. You might end the session, correct the bug using the editor and then retest the program. You also might step through the remainder of the program before ending the debugging session. To step through the rest of the program simply continue to press the F7 key until the highlight reaches the last line of the program, the one with the final `end`. When the highlight is on the last line of the program, all you need to do to end the debugger session is press the F7 key one last time.

*ending a  
session*

The Watch window will not clear automatically, so when you are finished with the debugger, you would normally clear it using the **Remove all watch** command on the **Break watch** menu.

*clearing the  
Watch window*

There are a few items to keep in mind when using the debugger. The debugger is line oriented. It executes one line at a time. If there are two (or more) statements on a line, then when the highlight is moved from that line, all the statements on the line will be executed. If a statement covers more than one line, it will still be executed as the highlight moves over it.

While in a debugging session, it is easy to find yourself in the editor and to accidentally change the program text. If this happens, use the editor to restore the text to look as it did before you changed it. When you resume the debugging session a window will ask "Source modified, rebuild? (Y/N)." Answer by pressing the N key and resume the session. If the program text is not exactly as before, then the highlight will be in the wrong location. If instead of answering "no" by pressing the N key, you were to press Y for "yes", then a new session will begin. The system will recompile your (possibly changed) program and start over.

---

## Breakpoints

You can mark lines of your program as *breakpoints*. These breakpoints are lines that are marked as stop points. When you run a program with breakpoints, program execution stops at the first breakpoint. The Watch window will show the values of all the expressions you have placed in that window. From that point you can step through the rest of your program using the F7 key exactly as we described in the previous section. If you want to see your program moving faster, you can restart the program. To restart

---



*setting  
breakpoints*

the program choose **Run** on the **Run** menu or use **Ctrl-F9**, just as you would do to run a program in the normal way. The program will then run until it reaches the next breakpoint or the end of the program, whichever comes first.

To set a breakpoint enter the editor and move the cursor to the line you want to mark as a breakpoint. Then choose **Toggle breakpoint** on the **Break watch** menu (or use the equivalent hotkey **Ctrl-F8**.) The line will be highlighted to indicate that it is marked. You may mark any number of lines as breakpoints. To remove a breakpoint do the exact same thing: move the cursor to the marked breakpoint and choose **Toggle breakpoint** on the **Break watch** menu (equivalently press **Ctrl-F8**.) The highlight will disappear indicating that the line is no longer a breakpoint. To remove all breakpoints choose **Clear all breakpoints** on the **Break watch** menu.

---

## Procedures

The debugger works exactly the same for programs with procedures as it does for simple programs without procedures. As you use the **F7** key to step through your program, the highlight will enter the text of a procedure declaration when the procedure is called. You can then step through the procedure using the **F7** key. When the procedure call ends, the highlight will return to the main body of the program. If you would like to suppress this detail and have the debugger treat a procedure call as if it were a single statement, use the **F8** key instead of the **F7** key. With the **F8** key the highlight goes directly from the procedure call to the next line in the main body of the program. The highlight never enters the procedure declaration. The procedure is still executed, but the debugger does not show you all the details. It only shows the result of the entire procedure call. If you wish, you can intermix using the **F7** and **F8** keys in the same session.

---

## Loops

Loops are treated just like any other kind of statement by the debugger, but this can sometimes give the impression that the debugger has stopped. For example consider the following *for* loop:

```
for I := 1 to 100 do
  Sum := Sum + I/2
```

When the debugger reaches the line that begins with **Sum**, you will have to press the **F7** key one hundred times before it moves on to the next line. This is because that line is executed one hundred times. In order to see that something really is happening, add the variables **I** and **Sum** to the **Watch** window.

Of course, you do not usually want to step through a loop body one hundred times. One way to deal with this sort of problem is to place breakpoints before and after the loop. Then the program stops before the loop. If you then restart the program, it will



run to the breakpoint after the loop. From that point you can step through the program or do whatever else you want.

Another alternative for dealing with long loops is to place an *if-then* or *if-then-else* statement inside of the loop so that you can mark one of the branches as a breakpoint. That way the program will stop during some, but not all, loop iterations. For example, you might trace the above loop by changing the loop to the following and marking the line that contains `Break` as a breakpoint.

```
for I := 1 to 100 do
  begin
    Sum := Sum + I/2;
    if (I mod 10) = 0 then
      Break
  end
```

`Break` can be any statement. It is just there so that you have something to mark as a breakpoint. You might declare a procedure called `Break` that does nothing, or you might replace `Break` with some “do nothing” statement such as

```
Sum := Sum
```

Unfortunately, the empty statement cannot normally be marked so that the debugger stops on it. You will probably need to write something that looks like a statement on the breakpoint line. You may need to try a few different alternatives before you find the exact technique that works best for you.

---

## Arrays

The debugger treats an array just as it treats any other variable. To add an array variable to the Watch window bring up the Add Watch window just as you would for any other variable. (Pressing `Ctrl-F7` is one way to do this.) Type the array name into the window, without any square brackets or indexes, and press return. The value of the array will be displayed as a list of values showing the values of the individual indexed variables.

---

## Units

The debugger handles units much as it handles procedures. When stepping through a program, the debugger brings in a unit and displays it when the program uses something declared in the unit. Units must be compiled with the correct options set so that the debugger has the all the information it needs. If the debugger works on programs without units, but it does not enter a unit correctly, recompiling the unit should correct the problem.

---

## ***Appendix 19***

# ***ASCII Character Set***

The following character numbers are used in TURBO Pascal. The numbers give the ordering of the type `char` as an ordinal type. For example, `chr (65)` will return the value 'A'. Only characters numbered 32 through 126 are printable. The remaining characters are nonprintable control characters.

<i>n</i>	chr ( <i>n</i> )	<i>n</i>	chr ( <i>n</i> )	<i>n</i>	chr ( <i>n</i> )	<i>n</i>	chr ( <i>n</i> )
0	^@ NUL	32	□	64	@	96	`
1	^A SOH	33	!	65	A	97	a
2	^B STX	34	"	66	B	98	b
3	^C ETX	35	#	67	C	99	c
4	^D EOT	36	\$	68	D	100	d
5	^E ENQ	37	%	69	E	101	e
6	^F ACK	38	&	70	F	102	f
7	^G BEL	39	'	71	G	103	g
8	^H BS	40	(	72	H	104	h
9	^I HT	41	)	73	I	105	i
10	^J LF	42	*	74	J	106	j
11	^K VT	43	+	75	K	107	k
12	^L FF	44	,	76	L	108	l
13	^M CR	45	-	77	M	109	m
14	^N SO	46	.	78	N	110	n
15	^O SI	47	/	79	O	111	o
16	^P DLE	48	0	80	P	112	p
17	^Q DC1	49	1	81	Q	113	q
18	^R DC2	50	2	82	R	114	r
19	^S DC3	51	3	83	S	115	s
20	^T DC4	52	4	84	T	116	t
21	^U NAK	53	5	85	U	117	u
22	^V SYN	54	6	86	V	118	v
23	^W ETB	55	7	87	W	119	w
24	^X CAN	56	8	88	X	120	x
25	^Y EM	57	9	89	Y	121	y
26	^Z SUB	58	:	90	Z	122	z
27	^[ ESC	59	;	91	[	123	{
28	^\ FS	60	<	92	\	124	
29	] GS	61	=	93	]	125	}
30	^ RS	62	>	94	^	126	~
31	^_ US	63	?	95	_	127	DEL



# Glossary

**actual parameter** A variable or expression that is substituted for a formal parameter when a procedure or function is called. See *value parameter* and *variable parameter* for an explanation of the two forms of substitution in Pascal.

**address** The address of a memory location is a number used as a way of naming that location. The memory locations are numbered 0, 1, 2, and so forth.

**algorithm** Detailed, unambiguous, step-by-step instructions for carrying out a task.

**alphanumeric character set** A set of symbols consisting of letters, digits, and special symbols.

**argument** Another name for *parameter*, particularly an actual parameter to a function.

**ASCII** The American Standard Code for Information Interchange. An arbitrary assignment of numbers to displayable characters and control signals.

**assembly language** Almost the same thing as machine language. The only difference is that assembly language instructions are expressed in a slightly more readable form, instead of being coded as strings of zeros and ones. See *machine language*.

**assertion** A comment that, if the program is correct, will be true when the program execution reaches the assertion.

**auxiliary storage** Another name for *secondary memory*.

**batch processing** As used in this book, running a program in batch mode means entering the program and all its data at one time and waiting until the program finishes before receiving the output.

**binary search** A search technique for finding the location of a value in an ordered list or other structure for keeping ordered data values, such as a binary tree. The list is divided in half, and the search proceeds to search the half that is a candidate for the sought value, that half is divided in half to obtain one quarter of the list, and so forth.

**binary tree** A tree in which each node has at most two descendants. See *tree*.

**bit** A binary digit; that is, a digit that may be either zero or one but can assume no other value.

**block** A set of statements enclosed with *begin* and *end* and separated by semicolons, together with the declarations and parameters that apply to them. Applied to programs and procedures.

---

**boolean expression** An expression that evaluates to either *true* or *false*. Used, for example, in *if-then-else* statements.

**bottom-up** Applied to any method that goes from subparts to larger parts made up of the subparts. See *bottom-up testing* in the text for an example of usage.

**buffer** A location for holding data that is on its way from one place to another.

**bug** A mistake in a program.

**byte** 1. Eight bits. 2. A location, or part of a location, in memory that holds eight bits.

**call** The interruption of a program's execution in order to execute a procedure or function.

**call-by-name parameter** Approximately the same as a *variable parameter*.

**call-by-value parameter** A synonym for *value parameter*.

**central processing unit** See *CPU*.

**code** Sometimes used to mean a program or part of a program.

**compile** 1. To translate a program using a compiler. 2. The action of a compiler.

**compiler** A program that translates programs from a high level language to machine language.

**compile-time error** An error discovered by the compiler, typically a syntax error.

**component** One unit of data stored in a file or other structured type value.

**component variable** 1. Of a record variable: The variable obtained by specializing the record variable to a single field; the syntax is to append a period followed by the field name. 2. Of an array: The same thing as an *indexed variable*.

**constant** In a Pascal program: the name of a value. It may be a number constant, such as 5 or 3.5, or a quoted string or character. It may also be the name associated with one of these constants by a constant declaration.

**control variable** A variable of an ordinal type that is used to control a *for* loop.

**CPU** The central processing unit of a computer. It performs the actual calculations and the manipulation of memory according to the instructions in a machine language program.

**crash** For a computer to suddenly stop working, often with disastrous results to the programs that are running at the time of the crash.

**cursor** A mark on a display screen that indicates where writing or other actions will take place; typically, a rectangle or small line of light.

**data** 1. The input to an algorithm or program. 2. Any information that is available to an algorithm or to a computer.

**data structure** A structure for holding related data values in an organized way; examples are arrays, records, lists, and trees.

**disk** A secondary memory device consisting of a disk that stores information in concentric tracks. There are two common types of disk: hard disks are semipermanently

mounted and relatively large; floppy disks are smaller and are remounted at each session.

**EBCDIC** Extended Binary Coded Decimal Interchange Code. An arbitrary assignment of numbers to display characters and control signals.

**echoing** Displaying the data typed in at the keyboard on the display screen.

**editor** A program that allows the computer to be used as a typewriter. An editor also has a number of commands that are more powerful than those of a typewriter, such as a command that moves an entire piece of text from one place to another.

**efficiency** The efficiency of a program is measured by the amount of resources that the program consumes. The less resources it consumes, the more efficient it is. Time and storage are the resources that are usually considered.

**element** A single data item, typically of an array or set.

**empty statement** In Pascal: a statement written by writing nothing and which does nothing when executed. Its main purpose is to allow the insertion of extra semicolons.

**enumerated type** A user-defined type consisting of a list of identifiers.

**execute** When an instruction is carried out by a computer, either directly or in some translated form, the computer is said to execute the instruction. When a computer follows the instructions in a complete program, it is said to execute the program.

**expression** In Pascal: any representation of a value, such as a variable, constant, arithmetic expression or boolean expression.

**field width** In Pascal: a specification that tells how many spaces to leave for an argument to `writeln` or `writeln`. It consists of a colon followed by an integer expression. For values of type `real`, a second colon and integer expression may be used to specify the number of digits to output after the decimal point.

**file** A named collection of data stored in secondary memory.

**file manager** Also sometimes called a *filer* or *file system*. A program that allows the user to store and retrieve objects called *files*. Among other things, a file can contain a Pascal program. Hence, a file manager is the program used to store and retrieve Pascal programs.

**flag** Any device that changes value to indicate that some event has taken place; boolean variables are often used as flags in Pascal programs.

**floating point numbers** In Pascal: Numbers of type `real`, especially when written in the E notation.

**formal parameter** An identifier in a procedure or function declaration heading that is changed when the procedure is called. See *value parameter* and *variable parameter*.

**friendly** A program is said to be *friendly* if it is easy to interact with, usually because it is very free about the format for entering data.

**fully exercise** A technique for testing a piece of code (like a program or procedure). It consists of finding a set of test inputs such that running the program on the test inputs will cause each statement and substatement to be executed on at least one of the test

---



runs and will cause each boolean expression that controls a loop to assume the value `true` on at least one run and `false` on at least one run.

**garbage collection** Making no-longer-needed memory locations available for reuse. In standard Pascal this is done using `dispose`.

**global variable** A variable declared in the main block of a program.

**hard copy** Output on paper, as opposed to output that goes to a file or other electronic means of storage.

**hardware** The actual, physical parts of a computer or computer system.

**hexadecimal numeral** A base sixteen numeral.

**high level language** A programming language that includes larger, more powerful instructions and, typically, a grammar that is somewhat like English. Programs in a high level language usually cannot be directly executed by computers. See *machine language*.

**implementation-dependent** A language detail that is different for different implementations; typically, it is not fully specified in the definition of the language, but left up to the implementer to implement in any way that is convenient and reasonable.

**index** The value that specializes an array, such 5 in `A[5]`.

**index variable** An array variable specialized to one component, such as `A[5]` or `A[I]`.

**initialize** Giving the first value to a variable in a program.

**intelligent** Having properties similar to a computer. For example, an intelligent terminal can do many computer functions without being connected to a computer.

**interactive** An interactive computer system is one that converses with the user via a terminal.

**invalid index** An error condition that occurs when a program attempts either to refer to an array location that is outside the declared range or to refer to a subrange type value that is outside the subrange declared for that type.

**invoke a function or procedure** To *call* a function or procedure.

**iterate** To repeat. Often used to mean an execution of the body of a loop.

**K** Abbreviation of *kilo*; originally it stood for 1000. When referring to computers, it now stands for 1024 (the power of 2 nearest to 1000). For example, 64K bytes means 65,536 bytes.

**kilo** See *K*.

**line printer** A typewriterlike output device that writes an entire line of text at one time.

**listing** 1. A copy of a program or the contents of a text file written on paper by some computer output device. 2. A copy of a program in a text file.

**local identifier** An identifier that is declared within a procedure or function, such as a local variable or local constant.

---

**local variable** A variable declared within a procedure or function.

**logical error** A program error that is due to an error in the algorithm or an error in translating the algorithm into the programming language. Normally, logical errors produce no error messages.

**machine language** A language that can be directly executed by a computer. Programs in machine language consist of very simple instructions, such as “add two numbers.” These simple instructions are coded as strings of zeros and ones. See *assembly language*.

**main memory** The memory that the computer uses as temporary “scratch paper” when actually carrying out a computation. See *secondary memory*.

**matrix** 1. A two-dimensional arrangement of numbers. 2. A two-dimensional array.

**mega-** A prefix meaning one million; for example, a megabyte is one million bytes.

**microcomputer** A small computer, usually for the use of a single person.

**minicomputer** A medium-size computer, typically shared by a number of users simultaneously.

**mnemonic** A memory aid. It frequently refers to the spelling of identifiers so as to hint of their use.

**modem** A device for transmitting digital information (typically to a computer) over a line such as a telephone line.

**modularity** A modular program is one that is divided into units (such as procedures) in such a way that each unit has well-defined means of communicating with the rest of the program (such as via parameters).

**monitor** 1. A display screen. 2. A program to trace another program.

**nano-** A prefix meaning one billionth; for example, a nanosecond is one billionth of a second.

**nesting** Placing one unit inside another unit. For example, nesting a smaller statement inside an *if-then* statement.

**object program** The translated version of a program produced by a compiler is called the *object program*. See *source program*.

**octal numeral** A base eight numeral.

**operating system** A program that is part of the system software of a computer. It is the program that controls and manages all other programs.

**operator** In Pascal: a symbol, such as + or \*, or certain identifiers, such as *div*, that are used to combine two values and produce a third value.

**ordinal type** A type that is considered to be an ordered list of values. In Pascal, the only ordinal types are *integer*, *char*, *boolean*, *enumerated types*, and *sub-range types*.

**overflow** The condition that results when a program attempts to compute a numeric value that is larger than the largest value of that type which the computer can represent

---

in memory, or is smaller than the smallest value of that type which the computer can represent in memory. See also *stack overflow*.

**parameters** The mechanism for passing information to or accepting information from a procedure or function. See *formal parameter*.

**pass a parameter** The value or name of an actual parameter is said to be *passed* when the procedure (or function) is called.

**peripheral device** An input, output, or secondary storage device of a computer.

**pop** To remove an element from a stack.

**portability** A program is *portable* if it can be moved from one system to another with little or no change.

**precedence** If an expression contains several different operations, the precedence is the order in which the operations are performed.

**program** An algorithm that a computer can either follow directly or translate and then follow the translated version.

**prompt line** A line of output telling the user to enter input.

**pseudocode** A mixture of English and Pascal (or some other combination of a natural language and a programming language).

**pseudorandom number** A number produced by a function or procedure designed to enumerate random-looking numbers.

**push** To add an element to a stack.

**radian** A unit for measuring angles. There are  $2\pi$  radians in 360 degrees.

**range checking** Automatic checking of variable or index values to see if they lie within declared subranges.

**reference parameter** Another term for *variable parameter*.

**reserved word** An identifier whose meaning is defined by the specification of the Pascal language and whose meaning cannot be changed by the programmer.

**root** See *tree*.

**running a program** When a program and some data are given to a computer in such a way that the computer is instructed to carry out the program using the data, that is called running the program (on the data).

**run-time error** A program error that is discovered by the computer system at the time the program is run. See *syntax error*.

**scalar type** Sometimes used as another term for *ordinal type*. Sometimes used to mean a type that is either an ordinal type or the type `real`.

**secondary memory** The memory a computer uses to store information in a permanent or semipermanent state. (When the computer does not have sufficient main memory for a computation, then it is also used as an addition to main memory.) See *main memory*.

**semantics** The meaning of a program or program construct or part of a program.

---



**sentinel value** A value in an array, file, input list, or other structure that marks an end of the structure.

**side effect** Something done by a procedure or function, other than changing an actual variable parameter or returning a function value, that has an effect beyond the procedure or function itself. Changing a global variable is an example of a side effect.

**simple type** A type that is either an ordinal type or the type *real*.

**software** Another term for *programs*.

**source program** The input program to be translated by a compiler is called the *source program*. See *object program*.

**stack** A particular data structure used, among other things, for keeping track of recursion. See the index for the location of the section that explains this.

**stack overflow** An error condition that results when a data structure called a *stack* is given more data than it can hold. It is a run-time error. One likely cause is infinite recursion.

**standard version of a programming language** A version of a programming language that is defined by some official standards organization.

**statement** The unit of action in a Pascal program. Statements are usually separated by semicolons. See the syntax diagrams in Appendix 4.

**step-wise refinement** A method of developing a program by dividing the entire task into subtasks.

**string** 1. A string of characters. 2. A quoted string, as in a constant declaration. 3. A packed array of characters type. 4. Any type whose values are strings of characters.

**structured programming** A method of programming that includes step-wise refinement for designing programs and includes coding the program in a form consisting of well-defined modules in a hierarchical arrangement.

**structured type** A type whose values are made up of other, simpler values, such as an array, file, or record type.

**subprogram** A portion of a program treated as a unit and defined outside the main body of the program. In Pascal, the word *procedure* is usually used rather than *subprogram* to mean the same thing.

**subroutine** A synonym for *subprogram*.

**subscript** Of an array: the same thing as an index.

**subscripted variable** An indexed variable of an array.

**syntax** The grammar rules of a language. The syntax rules of Pascal tell which strings of symbols are allowed as programs, statements, and so forth.

**syntax error** An error consisting of a violation of the syntax rules of a language. Syntax errors are discovered and reported by the compiler. See *run-time error*.

**system software** Refers collectively to all the programs that handle user programs. Included under this heading are such programs as *compilers* and *operating systems*.

---

**terminal** A device for communication with a computer, typically consisting of a keyboard and a display screen.

**testing all paths** A technique for testing a piece of code (such as a program or procedure). It consists of finding a set of test inputs such that running the program on the test inputs will cause each possible combination of branch and loop behaviors to occur on at least one of the runs.

**top-down** Any method that goes from the entire task or program to subparts, then the sub-subparts, and so on. For example, *top-down design* means the same as *step-wise refinement*.

**tracing** Inserting `writeln` statements into a program so that the values of the variables will be written out as the program performs its calculations. Some systems have debugging facilities that do this automatically.

**tree** A data structure consisting of nodes connected by pointers and satisfying certain conditions. Informally, these conditions require that, when drawn on paper, the structure exhibit a branching structure similar to a tree or upside-down tree. The nodes pointed to by a given node are called its “descendant” or “children” nodes. There is one node called the “root” node that has the property that any other node can be reached from it by following pointers.

**truth table** A table showing the values of a boolean expression for all possible argument values.

**type clash** A mismatch of types in the assignment of expressions to variables or in actual parameters of a procedure call or anyplace else.

**underflow** The condition that results when a program attempts to compute a value of type `real` such that the value is smaller in absolute value than the smallest positive `real` value that the system can represent in memory. In other words, the condition that results when a program attempts to compute a nonzero value that is too close to zero to be represented in memory (except possibly by the approximately equal value of zero).

**value parameter** Formal and actual parameters come in pairs. The pair is either a pair of value parameters or a pair of variable parameters. If the formal parameter is not prefaced by `var` in the formal parameter list of the procedure heading, then the pair is a pair of value parameters. If the pair is a pair of value parameters, the actual parameter may be anything that evaluates to the type of the corresponding formal parameter. A formal value parameter is a local variable. When the procedure is called, the value of the formal value parameter is initialized to the value of the corresponding actual value parameter.

**variable parameter** Formal and actual parameters come in pairs. The pair is either a pair of value parameters or a pair of variable parameters. If the pair is a pair of variable parameters, then a `var` is written in front of the formal parameter in the formal parameter list of the procedure heading. If the pair is a pair of variable parameters, the actual parameter must be a variable. A formal variable parameter is a labeled blank. When the procedure is called, the actual variable parameter is substituted for the corresponding formal variable parameter.

---

**verification** Verifying a program means proving that it meets the specifications for the task it is supposed to perform.

**volatile memory** Memory that loses its data contents when the power is shut off or when a program is finished running.

**word** 1. A location in main memory. 2. The contents of a location in main memory.

**word size** The number of bits that make up a word or memory location.

---



# Answers to Self-Test Exercises

## Chapter 1

### 1. Algorithm to add two whole numbers:

*begin*

1. Write the two numbers down one above the other so that they line up digit by digit with the rightmost digits one above the other; (If the two numbers are not of the same length, then add extra zeros to the front of the shorter number until they are of equal length.)
2. Add the two rightmost digits, obtaining a one- or two-digit number;
3. Write the rightmost of these two digits down as the rightmost digit of the answer and remember the leftmost digit; Call the digit that needs to be remembered by the name Carry; (If the number has only one digit, then the new value of Carry is zero.)
4. Do 4a, 4b and 4c again and again until you run out of digits:  
(If the two numbers are each only one digit long, then you "run out" before you start and so do 4a, 4b and 4c zero times, i.e., not at all.)
  - 4a. Move to the next pair of digits to the left;
  - 4b. Add these two digits and the Carry, obtaining a new one- or two-digit number;
  - 4c. Write the rightmost digit of the number so obtained as the next (reading right to left) digit of the answer and use the leftmost digit of this number as the new (possibly changed) value of Carry; (If the number has only one digit, then the new value of Carry is zero.)
5. If Carry is zero at this point (i.e., at the left end of the two numbers), then you are done;
6. If Carry is not zero, then write down the value of Carry as the leftmost digit of the answer

*end.*

### 2. As with virtually all problems, there is more than one algorithm for this problem. One algorithm is:

*begin*

1. Write the word down on one line;
2. Write it down on the line below, but this time write it backwards; (Align the letters on the two lines.)
3. For each letter in the word:  
compare the letter to the one written just below it;
4. If all letters matched then the word is a palindrome; if at least one mismatch was found, then it is not a palindrome

*end.*

3. This algorithm assumes that the input word is written on a sheet of paper.

*begin*

1. Write the letters of the alphabet down on a sheet of paper, one per line;
2. Write zero after each letter;
3. Place your finger on the first letter of the word;
4. Repeat the following until you run out of letters (at the end of the word):
  - 4a. Read the letter pointed to by your finger;
  - 4b. Add one to the number written after that letter on the sheet of paper; (The old number is erased or crossed out.)
  - 4c. Move your finger to the next letter in the word (provided there is one);
5. The number of occurrences of each letter is written on the sheet of paper

*end.*

## Chapter 2

1. 2 2

If your system outputs

22

(i.e., no blanks between the numbers), then on your system you need to explicitly insert a blank between any two consecutive numbers output. One way to do this is the following:

```
writeln(X, ' ', Y)
```

(The second argument is a blank in single quotes.)

2. The output is the single number:

3

3. BCBC

4. They are all *incorrect* except for the constant 4 (with no decimal point).

5. The following are all *incorrect*. The rest are correctly formed.

.89    -.89    3,987.85    4.    4

4, is a correctly formed constant of type `integer`, and so it can be used anywhere that a constant of type `real` can be used; that makes 4 pragmatically as good as correct.

6. The following are all *incorrect*. The rest are correctly formed.

.57E12    57E3.7    57.9E3.7

7.  $3 * X$      $3 * X + Y$      $(X + Y) / 7$   
 $(3 * X + Y) / (Z + 2)$

8. (a) In a sense it is correct as is. The compiler will accept it and process it properly. However, the style is very poor. The spacing and line breaks are not well designed for readability. (b) In `TURBO Pascal` this is allowed. In standard Pascal the identifier `var` should only be used once. (c) The first semicolon should be a comma. (d) It needs a semicolon at the end.

9. `integer    real    char`  
`real    real    real`

10. Type it up and compile it; the computer will tell you the first mistake as well as a guess of the other mistakes. Correct the first mistake, and compile it again. Continue until you have corrected all the mistakes. (The first mistake is that the first line needs a semicolon. The second mistake is that the double quotes are used where single quotes should be used. The other mistakes are all missing punctuation marks: a closing single quote, a semicolon and the final period.)
13. `15 div 12 is 1`  
`24 div 12 is 2`  
`123 div 100 is 1`  
`200 div 100 is 2`  
`99 div 2 is 49`  
`2 div 3 is 0`
- `15 mod 12 is 3`  
`24 mod 12 is 0`  
`123 mod 100 is 23`  
`200 mod 100 is 0`  
`99 mod 2 is 1`  
`2 mod 3 is 2`

### Chapter 3

1. `START -1234END`  
 (There are three spaces between the *T* and the minus sign.)
2. `START -12.35END`  
 (There are two spaces between the *T* and the minus sign.)
6. `Second writeln`  
`Fourth writeln`
7. `false false`  
`true true`
8. `4, 3, 6, -6, 7, -7, 6.8,`  
`6.8, 4, 4, 4`
9. `sqrt(x) <= y + 1`  
`Z > 0 W <> 0 X mod 12 = 0`
10. `sqr(X + (Y/(X + Z)) + W)`  
`or sqr(X + Y/(X + Z) + W)`  
`abs(X - Y)`  
`sqrt(((X + 3*Z)/W) + Y)`  
`or sqrt((X + 3*Z)/W + Y)`

### Chapter 4

1. `Begin Conversation`  
`Goodbye`  
`Hello`  
`One more time:`  
`Hello`  
`Goodbye`  
`End conversation`
2. `3 6`  
`6 3`
3. `One`  
`One Two`  
`One Two Three`



4. 1 2  
2 2  
2 1

5. `procedure NoNeg (var N: integer);`  
  `begin{NoNeg}`  
    `if N < 0 then`  
      `N := 0`  
    `end; {NoNeg}`

6. `procedure NoCapY (var Ans: char);`  
  `begin{NoCapY}`  
    `if Ans = 'Y' then`  
      `Ans := 'y'`  
    `end; {NoCapY}`

7. *Yes*, a *variable* parameter can be used to give information to a procedure. *Yes*, a *value* parameter can be used to give information to a procedure. *Yes*, a *variable* parameter can be used to get information out of a procedure. *No*, a *value* parameter cannot be used to get information out of a procedure.

8. They are all allowed as actual value parameters. Only X is allowed as an actual variable parameter.

9. *Yes*, an actual *variable* parameter can be a variable. *Yes*, an actual *value* parameter can be a variable. *No*, an actual *variable* parameter cannot be a constant. *Yes*, an actual *value* parameter can be a constant.

10. 1 2  
2 1

## Chapter 5

1. 7  
11

2. Hi Folks in procedure  
21 outside of procedure

3. 1 2 3  
4 5 6  
4 5 3

4. 1 2 3  
4 5 6  
5 4 3

5. 1 2 3  
5 5 6  
5 2 3

6. 1 2 3  
4 5 6  
4 2 5

Even though there is a local variable named C, the global variable C can be changed *provided it is an actual parameter*. When the procedure is executed there are two distinct variables

---

named C, one global and one local. (If you want a more detailed explanation of how this can be done, you should consult the optional section of Chapter 4, entitled "Implementation of Variable Parameters.")

7. 1 2 3  
4 5 6  
1 2 3

8. XY  
AB  
XB

## Chapter 6

### 1. Start

First writeln  
Next  
Enough

2. false                    true  
   true                    false  
         false  
         false  
         true

### 3. $2 + 2 = 4$

$(X + 7 > 100) \text{ or } (X + 7 < 50)$   
 $(Z \neq 'A') \text{ and } (Z \neq 'B') \text{ and } (Z \neq 'C')$   
 -The following is also correct, although a bit harder to read:  
 $\text{not} ( (Z = 'A') \text{ or } (Z = 'B') \text{ or } (Z = 'C') )$   
 $(X \bmod 3) \neq 0$   
 -It is all right to omit the parentheses around  $X \bmod 3$ -  
 $(X \bmod 3 \neq 0) \text{ or } (Y \bmod 5 = 0)$   
 $(X \leq Y + 2) \text{ and } (Y + 2 \leq Z)$

### 4. $(A \text{ or } B) \text{ and not}(A \text{ and } B)$

There are other equivalent expressions that would be correct here. For example, the following will also work:

$(A \text{ and not } B) \text{ or } (B \text{ and not } A)$

### 5. First writeln

Fourth writeln

### 6. Program Exercise(input, output);

```
var N1, N2, N3: integer;
begin{Program}
  writeln('Enter three integers:');
  readln(N1, N2, N3);
  if (N1 <= N2) and (N2 <= N3) then
    writeln('In order.')
  else
    writeln('Not in order.');
```

writeln('End program')

```
end. {Program}
```

7. 

```
if X < 0 then
    writeln(X, ' is Negative.')
else if (0 <= X) and (X <= 100)
    then writeln(X, ' is between 0 and 100.')
else { X > 100}
    writeln(X, ' is greater than 100.')
```
8. The types `integer`, `char`, and `boolean` may be used; the types `real` and `string` may not be used. Use of the type `boolean` is pointless since an *if-then-else* statement is a preferable way to accomplish the same effect, but the type `boolean` is legal.
9. 

```
program Showcase(input, output);
var MonthNum: integer;
begin{Program}
    writeln('Enter a month as a number between 1 and 12. ');
    writeln('I'll tell you how many days it has in it. ');
    readln(MonthNum);
    case MonthNum of
        4, 6, 9, 11: writeln('30 days');
        1, 3, 5, 7, 8, 10, 12: writeln('31 days');
        2: writeln('28 days (29 if leap year)');
    end; {case}
    writeln('That's it !')
end. {Program}
```
10. *not* (`FootLoose`) and *not* (`FancyFree`) is equivalent to (always evaluates to the same value) as: *not* (`FootLoose or FancyFree`)  
The other two expressions are equivalent to each other, but not equivalent to the above two.

## Chapter 7

1. -2
2. The boolean expression evaluates to `false` and so the loop body is never executed. The output is thus the number 10.
3. This is an infinite loop. There is no regular output, although most systems will eventually produce an error message after the program consumes too much time or when the program attempts to increase the value of `X` above `maxint`.
4. There are two mistakes: (1) The value 1 is never output. (2) The value of `X` is never exactly 10 and so there is an infinite loop. One way to correct the code is:

```
X := 1;
while X < 10 do
    begin{while}
        write(X);
        X := X + 2
    end {while}
```



5. -2

6. The body of a *repeat* loop is always executed at least once; the body of a *while* loop may be executed zero times.

7. *while* <boolean> *do* <statement>

is equivalent to:

```
if <boolean> then
  repeat
    <statement>
  until not (<boolean>)
```

```
repeat <body> until <boolean>
```

is equivalent to:

```
begin
  <body>
end ;
while not (<boolean>) do
  begin
    <body>
  end
```

```
8. program EchoLetter(input, output);
   var Symbol: char;
   begin{Program}
     writeln('Enter a line of text. ');
     writeln('End by pressing the return key. ');
     while not eoln do
       read(Symbol);
       writeln('The last symbol you typed in is ', Symbol)
   end. {Program}
```

Alternatively, the *while* loop may be replaced by the following:

```
repeat
  read(Symbol)
until eoln
```

9. -6 -4 -2 0 2 4 6 8 10 12 14 16 18 20 22

10. 10 9 8 7 6 5 4 3 2 1

```
11. program Even(input, output);
   var I: integer;
   begin{Program}
     for I := 1 to 12 do
       write(2 * I);
     writeln;
     writeln('That''s all folks!')
   end. {Program}
```

12. 1 3  
2 2  
3 2

13. The output is too long to reproduce here. The pattern is indicated below:

```

1 times 10 equals 10
2 times 10 equals 20
.
.
.

10 times 10 equals 100
1 times 9 equals 9
2 times 9 equals 18
.
.
.

10 times 9 equals 90
1 times 8 equals 8
.
.
.
```

14. a. A *for* loop.  
 b. and c. Both require a *while* loop since the input lists might be empty.  
 d. A *repeat* loop can be used since some nonzero number of tests will be performed.

## Chapter 8

1. ABX
2. The function name, `TwoPower`, is not a variable but is being used as a variable. The following is not allowed:  

```
TwoPower := TwoPower*2
```
3. 

```
function Area (Length, Width: real): real;
{Returns the area of a rectangle of the given dimensions.}
begin{Area}
    Area := Length * Width
end; {Area}
```
4. 

```
function Pos (I: integer): char;
begin{Pos}
    if I > 0 then
        Pos := 'P'
    else
        Pos := 'N'
end; {Pos}
```
5. a. A function.  
 b. Since it computes two values, a procedure with one value and two variable parameters should be used. (Two functions could be used, but since the two computations are so interrelated they do not separate neatly into two subtasks, a procedure is preferable.)  
 c. A procedure.  
 d. Since it computes two values, it makes sense to use a procedure with two variable parameters (in addition to two value parameters). Alternatively, two functions could be used: one to compute net income from gross income and adjustments and another to compute the tax from the net income.  
 e. A function.

6. Hi  
Good-Bye
7. *function* Divides(A, B: integer): boolean;  
 {Returns true if A evenly divides B; otherwise, returns false.  
 Precondition: A is not zero.}  
*begin*{Divides}  
     Divides := (B mod A = 0)  
*end*; {Divides}
- The following also works, but is poor style:
- function* Divides(A, B: integer): boolean;  
 {Returns true if A evenly divides B; otherwise, returns false.}  
*begin*{Divides}  
     if (B mod A = 0) then  
         Divides := true  
     else  
         Divides := false  
*end*; {Divides}
8. *function* InOrder(A1, A2, A3: integer): boolean;  
 {Returns true if A1 <= A2 <= A3; otherwise returns false.}  
*begin*{InOrder}  
     InOrder := (A1 <= A2) and (A2 <= A3)  
*end*; {InOrder}
9. *function* Ran2to20(var Memory: integer): integer;  
 {Returns a pseudorandom even number between 2  
 and 20, inclusive. Uses the function Random in Figure 8.9.}  
 var OneToTen: integer;  
*begin*{Ran2to20}  
     OneToTen := (Random(Memory) mod 10) + 1;  
     {OneToTen is a pseudorandom number between 1 and 10.}  
     Ran2to20 := 2 \* OneToTen  
*end*; {Ran2to20}

10. SmallNegNumber is illegal because -1 is greater than -100. GradePoint is illegal because you can not have a subrange of the type real. SmallRange is illegal because it contains an expression (other than a constant) in its definition. All the other type definitions are legal.
11. *type* Score = 0 .. 100;  
     NonNegInteger = 0 .. maxint;  
     Abs100 = -100 .. 100;
12. No. There are no provisions in Pascal to either read in a value of an enumerated type or to write one out. A user defined procedure could be written to do something similar by reading or writing a string which names the value.

## Chapter 9

1. TempCount and GradeTally are illegal because the type real is not an ordinal type and because it can not have subrange types. All the others are legal.



```
2. type Score = 0 .. 10;
   ScoreList = array[1 .. 100] of Score;
   RealList = array['a' .. 'z'] of real;
   LastType = array[-5 .. 19] of char;
```

There are other ways of stating the definitions, for example, the following is also acceptable:

```
type Score = 0 .. 10;
   Index = 1 .. 100;
   ScoreList = array[Index] of Score;
   Alphabet = 'a' .. 'z';
   RealList = array[Alphabet] of real;
   LastType = array[-5 .. 19] of char;
```

```
3. a. type Score = 0 .. 10;
     Index = 1 .. 100;
     ExtendedIndex = 0 .. 100;
     ScoreList = array[Index] of Score;

     var A: ScoreList; {holds the scores.}
     Last: ExtendedIndex; {Tells how many are in the array.
     If the array is empty it is set equal to 0.}

     b. type TallyList = array['A' .. 'Z'] of 0 .. maxint;
        var Count: TallyList;
```

For example, Count['M'] will hold the number of students whose last name starts with 'M'. There are other variations that are correct. For example, it would be acceptable to use integer as the component type of the array.

```
c. type Checkoff = array[1 .. 100] of boolean;
   var Passed: Checkoff;
```

For example, if Passed[7] is true, then student number seven can graduate.

4. As written it sums the numbers between A[1] and A[100]. If the value of A[1] is 1 and the value of A[100] is 2, then the final value of Sum will be 1 + 2 or 3. The correct code is:

```
Sum := 0;
for I := 1 to 100 do
  Sum := Sum + A[I]
```

The variable I should be of type integer or better still type 1 .. 100.

```
5. for I := 0 to 100 do
   C[I] := 0.0
```

(It is permissible to use 0 in place of 0.0.)

6. C := 0 is not legal (nor is C := 0.0.) You must set the value of each array element separately.

7. The loop ends with the value of I equal to Last. At that point I + 1 evaluates to Last + 1, and so A[I + 1] has an illegal index. To fix it, change the final expression of the for loop to Last - 1.

```
8. for I := 1 to 6 do
   read(A[I]);
for I := 6 downto 1 do
  write(A[I]);
writeln
```

- 
9. 

```
Last := 0;
read (Next);
while (Next > 0) and (Last < 10) do
  begin
    Last := Last + 1;
    A[Last] := Next;
    read (Next)
  end;
if Next > 0 then
  writeln('Warning: some numbers are not in the array.');
```

```
for I := 1 to Last do
  write(A[I]);
writeln
```

10. 

```
const FirstNum = 661;
      LastNum = 753;
type Grade = 1 .. 10;
   Distribution = array[Grade] of integer; {part a.}
   Age = 5 .. 13;
   Count = array[Age] of integer; {part b.}
   CheckAmount =
     array[FirstNum .. LastNum] of real; {part c.}
```

(There is more than one right answer to this question.)

11. 

```
type WeekDay = (Mon, Tue, Wed, Thur, Fri, Sat, Sun);
   WorkDay = Mon .. Fri;
   WorkHours = array[WorkDay] of integer;
   PlayHour = array[WeekDay] of integer;
```

12. 

```
A := B
```

13. To start with the *if-then-else* statement is poor style. The preferable way to write it is:

```
Found := (N = A[I])
```

However, the real problem is that it often returns the wrong value. Try it with `Last` equal to 5 and with array elements:

```
2, 4, 6, 8, 10,
```

and with `N` set equal to 8. The code will decide to return `true` when it tests `A[4]` and finds its value is 8, but when it sees `A[5]` it changes its mind and incorrectly returns `false` as the final value. The code in Figure 9.10 shows the correct way to accomplish a similar task.

14. 

```
procedure Reverse(var A: Word);
var B: Word;
begin{Reverse}
  for I := 0 to 100 do
    B[I] := A[100 - I];
  A := B
end; {Reverse}
```

(It is possible to do this without the local variable `B`, but that is significantly harder to do.)

15. Replace the function `Imin` by a function `Imax` that computes the index of the largest element instead of the index of the smallest element.

---

```

function Imax (var A: List; Start, Index; Last: Index): Index;
{Returns the index I such that A[I] is the largest of the values: A[Start], A[Start + 1], . . . . A[Last].}
var Max, I: integer;
begin{Imax}
    Imax := Start; {tentatively}
    Max := A[Start]; {maximum so far}
    for I := Start + 1 to Last do
        if A[I] > Max then
            begin{then}
                Max := A[I];
                Imax := I
                {Max is the largest of the values A[Start], . . . . A[I];
                 the tentative value of Imax is x such that A[x] = Max}
            end {then}
    end; {Imax}

```

16. Change the array type and type of the parameters of Exchange, and the type of the local variable Min in Imin, all to real.

```

type . . .
    List = array[Index] of real;
procedure Exchange (var X, Y: real);
var Temp: real;
function Imin (var A: List; Start, Last: Index): Index;
var Min: real;
    I: integer;
    . . .

```

17. The same type changes as in Exercise 16, but use the type char rather than the type real.

## Chapter 10

1. In standard Pascal the array is not a legal argument for a writeln statement. TURBO Pascal does allow this and would output the string a b b b b.
2. The code for the procedure StringWrite is the same as that for the procedure StringWriteln except that the writeln is omitted.
3. 

```
const Last = 17;
      MaxLength = 20;
type CharString = array[1 . . MaxLength] of char;
      Index = 1 . . Last;
      NameList = array[Index] of CharString;
      BirthList = array[Index] of integer;
      VaccinationList = array[Index] of boolean;
```
4. Change the value of MaxLength to 40.
5. The first three are allowed. Some systems will and some systems will not allow the read statement. The last statement is an error because the string 'Hi!' is less than five characters long.
6. 

```
a. for I := 1 to 20 do
      writeln(S[I, 1])
```



```

b. for I := 1 to 50 do
    write(S[20, I])
c. for I := 1 to 20 do
    begin{line I}
        for J := 1 to 50 do
            write(S[I, J]);
        writeln
    end {line I}

```

## Chapter 11

1. 5    A  
6    A
2. type Sam =  
    record  
        Field1: integer;  
        Field2: real;  
        Field3: char  
    end;
3. program Exercise3(input, output);  
    const Space = ' ';  
    type Sam =  
        record  
            Field1: integer;  
            Field2: real;  
            Field3: char  
        end;  
    var X: Sam;  
    begin{Program}  
        writeln('Enter an integer:');  
        readln(X.Field1);  
        writeln('Enter a real:');  
        readln(X.Field2);  
        writeln('Enter a symbol:');  
        readln(X.Field3);  
        writeln('The record contains:');  
        writeln(X.Field1, Space, X.Field2, Space, X.Field3)  
    end. {Program}
4. type Student =  
    record  
        Name: array[1..20] of char;  
        QuizScores: array[1..10] of 0..10;  
        MidTerm, FinalExam, FinalScore: 0..100;  
        Grade: 'A'.. 'F'  
    end;
5. B[3]  
    B[3].Price  
    copy(B[10].Name, 6, 1)  
    length(B[2].Name)

## 6. Parallel arrays:

```
type StockNum = 0 .. Max;
   StyleNum = 0 .. 50;
   SizeRange = 3 .. 14;
   Style = array[StockNum] of StyleNum;
   Count = array[StockNum, SizeRange] of integer;
   Price = array[StockNum] of real;
```

Array of records:

```
type StockNum = 0 .. Max;
   StyleNum = 0 .. 50;
   SizeRange = 3 .. 14;
   Shoe = record
       Style: StyleNum;
       Count: array[SizeRange] of integer;
       Price: real
   end;
   InStock = array[StockNum] of Shoe;
```

Max is some defined constants.

7. [1, 3, 7, 8, 9], [7, 8, 9], [8], [ ], [7], true, false, true, true, false, true

## Chapter 12

1. The problem is a "boundary" problem. When I is equal to 100, the loop terminates without adding in the 100. The easiest way to fix it is to change the end of the repeat loop to:

```
until I > 100
```

2. The values of A and B, and C do not matter, but you need a collection of different values for X. You need one value of X that is greater than or equal to 5 and one that is less than 5. You also need one value greater than zero and one less than or equal to zero. For example, the following two values of X will do: 5, 0. Since some numbers satisfy more than two of the required cases, you do not need four test values. You should use other test values as well, but that is enough to fully exercise the code.
3. There are four paths, but one is impossible. One of many possible sets of values for X is: 5, 4, 0. The values of A and B, and C do not matter for testing all paths.
4. The program uses more storage than is needed for a clear program. There is no need to use an array. All that is needed is a single variable as shown below:

```
Sum := 0;
for I := 1 to 10 do
begin
    read(Next);
    Sum := Sum + Next;
end;
Average := Sum/10
```

The variable Next is of type integer. It is also a good idea to declare 10 as a named constant.

---

## Chapter 13

1. 5 63  
5 6

(Remember: the blank is a character.)

5. Four score and seven years ago,  
o  
t  
The End.
6. ab  
ghijkHi
7. *program* WriteTen; {TURBO but not standard Pascal}  
var NewFile: text;  
I: integer;  
*begin*{Program}  
assign(NewFile, 'NEW.TXT');  
rewrite(NewFile);  
writeln('Writing to NEW.TXT');  
for I := 1 to 10 do  
writeln(NewFile, I);  
close(NewFile);  
writeln('Done writing to NEW.TXT')  
*end.* {Program}
8. *program* ReadTen; {TURBO but not standard Pascal}  
var NewFile: text;  
Number, Sum: integer;  
*begin*{Program}  
assign(NewFile, 'NEW.TXT');  
reset(NewFile);  
Sum := 0;  
while not eof(NewFile) do  
*begin*{while}  
readln(NewFile, Number);  
Sum := Sum + Number  
*end;* {while}  
close(NewFile);  
writeln('The sum of the numbers is ', Sum)  
*end.* {Program}

## Chapter 14

1. 6
2. This is an example of infinite recursion. There will be no legitimate output. However, the error message *stack overflow* is likely.
3. 24  
The function is the factorial function, usually written  $n!$  and defined as  $n! = n*(n-1) * (n-2)* \dots *1$ .
4. 3  
(The function returns the value  $\log_2(N)$ , i.e., the logarithm to the base 2 of the number N, but you need not know that to see that the value returned is 3.).



## 5. Hip Hip Hurray

```

6. function Mystery(N: integer): integer;
   {Precondition:  $N \geq 1$ }
   var Sum, I: integer;
   begin{Mystery}
     Sum := 0;
     for I := 1 to N do
       Sum := Sum + I;
     Mystery := Sum
   end; {Mystery}

```

```

7. function Rose(N: integer): integer;
   {Precondition:  $N \geq 0$ }
   var Product, I: integer;
   begin{Rose}
     Product := 1;
     for I := 1 to N do
       Product := Product * I;
     Rose := Product
   end; {Rose}

```

```

8. procedure RecStar(N: integer);
   const Star = '*';
   {Writes N '*'s to the screen. Precondition:  $N > 0$ .}
   begin{RecStar}
     writeln(Star);
     if N > 1 then
       RecStar(N - 1)
   end; {RecStar}

```

You should also type up and run your procedure for this exercise.

## Chapter 15

1.	+	0	1	2	3	4	5	6
	-	0	1	2	3	4	5	6
	+	1	2	3	5	+	0	3
	-	1	2	3	5	+	0	3
	+	1	2	3	1	-	0	2
	-	1	2	3	1	-	0	2
	+	3	1	4	2	+	0	1

2. In each case, the largest value of type real is: +0.999E+999, +0.99E+9999, +0.99999E+9

3. 7, 5, 4, 27, 22, 0.5, 0.25, 0.125, 0.625, 1.125, 5.625

4. Thirty-two bit machine:

$$2^{31} - 1 = 0.2147836 \times 10^{10} \text{ (approx.)}$$

Sixty-four bit machine:

$$2^{63} - 1 = 0.9223372 \times 10^{19} \text{ (approx.)}$$

$$\begin{array}{r} 0.3000 \times 10^3 \\ + 0.00012345678 \times 10^3 \\ \hline 0.30012345678 \times 10^3 \end{array}$$

After rounding it is stored as  $0.3001 \times 10^3$ . Hence, the output is:

0.3001E+03

6. X\*Y\*Z has value

$$\begin{aligned} (1.0E-90 * 1.0E-25) * 1.0E+25 &= \\ (1.0E-115) * 1.0E+25 & \end{aligned}$$

but the first number produces floating point underflow and so is rounded to zero (on our hypothetical computer). Hence, the final value is  $0.0 * 1.0E+25$  which is equal to 0.0. On the other hand, X\*Z\*Y has value

$$\begin{aligned} (1.0E-90 * 1.0E+25) * 1.0E-25 &= \\ (1.0E-65) * 1.0E-25 &= 1.0E-90 = \\ 0.1000E-91 & \end{aligned}$$

Thus, the output is:

0.0      0.1000E-91

(As this shows, rounding to zero can occasionally produce surprising results.)

## Chapter 16

```
1. const MaxLength = 20;
   type Student =
       record
           Name: string[MaxLength];
           Final: 0 .. 100;
           Grade: 'A' .. 'F'
       end;
   GradeBook = file of Student;

2. program Writer; {TURBO but not standard Pascal}
   var NumFile: file of integer;
       I: integer;
   begin
       assign(NumFile, 'NUM.DAT');
       rewrite(NumFile);
       for I := 1 to 10 do
           write(NumFile, I);
       close(NumFile);
       writeln('End of Program')
   end.
```

- ```
3. program Reader; {TURBO but not standard Pascal}
   var NumFile: file of integer;
       I: integer;
   begin
       assign(NumFile, 'NUM.DAT');
       reset(NumFile);
       while not eof(NumFile) do
           begin
               read(NumFile, I);
               writeln(I)
           end;
       close(NumFile);
       writeln('End of Program')
   end.
```
- ```
4. program Searcher; {TURBO but not standard Pascal}
   var NumFile: file of integer;
       I, Key: integer;
       Found: boolean;
   begin
       writeln('Enter integer to be searched for');
       readln(Key);
       assign(NumFile, 'NUM.DAT');
       reset(NumFile);
       Found := false;
       while (not eof(NumFile)) and (not Found) do
           begin
               read(NumFile, I);
               if I = Key then
                   Found := true
               end;
           close(NumFile);
           if Found then
               writeln(Key, ' Found in file.')
           else
               writeln(Key, ' Not found in file.')
           end.
```
- ```
5. function Search(var F: PayRecords; N: integer): boolean;
   {Returns true if PayRecords contains a record
   with Number field equal to N; returns false otherwise.}
   var Found: boolean;
       OneRecord: Employee;
   begin {Search}
       reset(F);
       Found := false;
       while (not eof(F)) and (not Found) do
```
-



```

begin{while}
  read(F, OneRecord);
  if OneRecord.Number = N then
    Found := true
  end; {while}
Search := Found
end; {Search}

```

6. The following may be used only with text files:

readln, writeln, eoln, and seekln

The following may be used only with nontext files:

seek, filesize, and filepos

The following may be used with both text files and nontext files:

assign, reset, rewrite, read, write, eof,  
rename, erase, and close

## Chapter 17

1. 10 20

20 20

30 30

40 40

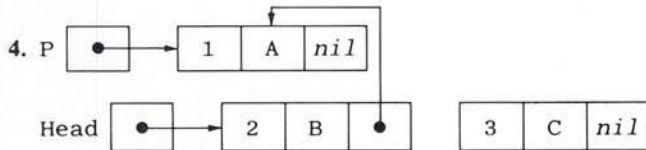
2. 10 20

20 20

30 20

30 40

3. It would not change at all.



5. type Pointer = ↑Node;

Node = record

Number: integer;

Link: Pointer

end;

procedure BuildList(N: integer; var Head: Pointer);

var PNew: Pointer;

I: integer;

```

begin{BuildList}
  Head := nil;
  for I := N downto 1 do
    begin{Insert I}
      new(PNew);
      PNew↑.Number := I;
      PNew↑.Link := Head;
      Head := PNew
    end {Insert I}
  end; {BuildList}

```

6. *procedure* ShowList(Head: Pointer);  
 var Next: Pointer;  
 begin{ShowList}  
   Next := Head;  
   while Next <> nil do  
     begin{Process one node}  
       writeln(Next↑.Number);  
       Next := Next↑.Link  
     end {Process one node}  
 end; {ShowList}

7. Replace the lines

```

Before↑.Link := PNew;
PNew↑.Link := After

```

with the lines:

```

PNew↑.Link := Before↑.Link;
Before↑.Link := PNew

```

Notice that the order of the last two assignment statements is now critically important. Also notice that if Before is pointing to the last node in the list, then, with this change, the procedure correctly inserts a node at the end of the list.

8. There is no difference in the general form of the nodes. The difference is in the way they are used.

```

9. type Pointer = ↑TreeNode;
    TreeNode = record
        Data: integer;
        Left, Right, Parent: Pointer
    end;

```

---

## Photo Credits

- P. 7 All photos: IBM Corporation
- P. 47 Left: IBM Corporation. Right: Smithsonian Institution
- P. 185 Trustees of the Science Museum, London
- P. 481 Left: Culver Pictures. Right: IBM Corporation. Bottom: IBM Corporation
-





---

# Index

- abs 93
  - Ada 10
  - Algorithm 8–9
    - partial 9
    - total 9
  - Al-Khowarizmi 8
  - Alt character A.36–37
  - Analogy (solution by) 58
  - and 183
    - short definition* 211
  - append, *short definition* 536
  - Argument *see* function, argument; parameter list
  - Arithmetic expressions 32–35
  - Array 331–417
    - short definitions* 367–368
    - of arrays 375–378
    - base type *see* array, component type
    - component type 333
    - constants 415–416
    - efficiency of variable parameter 352–353
    - element 334
    - empty 340
    - exceeding capacity 341
    - index 334
      - noninteger 354
      - with semantic content 355
    - indexed variable 333–334
      - short definition* 368
      - as a parameter 335
    - index out of range 341–343
    - index type 335–336
    - input/output 339–340
    - multidimensional 407–409
    - packed 404–405
    - packed array of characters 405–406
      - short definition* 417
    - parallel 379–380
    - parameter 338, 346–347, 350
    - partially filled 340–341
    - random access 355
    - of records 434–435
    - searching 348–352
    - as a structured type 346–348
    - type declaration 335–339, 409
      - short definition* 368, 417
    - variable 334, 346–348
  - Arrow 516
  - ASCII character set A.86–87
  - Ask before iterating 225
  - Assembly language 11
  - Assertion 73–74, 78–79
    - invariant 240–244
      - see also* postcondition; precondition
  - assign 508, 608
    - short definition* 534
  - Assignment operator 25–26
  - Assignment statement 25–27
  - Asterisk 26
  - Augusta, Ada 481
  - Automated Drill 389–394
  - Automobile bargaining 162–163
  - Averaging program 230
  - Babbage, Charles 481
  - BASIC 10
  - Batch processing *see* off-line data
  - B compiler directive A.73
  - Binary numerals 585–586
  - Binary search tree storage rule 681
  - Binary tree *see* tree
  - Bisection method 595
  - Bit 4, 586
  - Blackjack (scoring) 204–210
  - Blank 35, 37, 41, 46
    - editing out excess 526–529
    - padding with 375
    - in strings 395
  - Block 157–159
    - short definition* 175
  - Body *see specific kind, e.g.,* program, body
  - boolean 188, 293
    - short definition* 211
    - constant 193
    - input and output 191
  - Boolean expression 83–85, 182–185
    - short circuit evaluation A.73
    - undefined 186–187
  - Boole, George 185
  - Booting A.24
  - Bottom-up 165, 169
  - Boundary value 399, 488
  - Branching 81–84, 180
-

- multiple 202–203
  - see also* case statement; *if-then-else*
- Breakpoint A.83–84
- Brute force 349
- Buffer 522
- Bug *see* debugging
- Byte A.5, A.89
- byte A.5
- C 10
- Calendar 252–258, 303–307
- case 196–199
- Case statement 196–201
  - short definition*
  - else* clause 199
  - in TURBO Pascal 197–201
  - type of controlling expression 299
- cd A.32–33
- Change making 53–57, 128–132
- Change dir A.51–52
- Changing directory
  - in DOS *see* cd
  - in TURBO *see* Change dir
- char 29–30, 293
  - short definition* 60
- Character 29
  - nonprintable 296
  - see also* char
- checkeof A.62
- chkdsk A.30
- chr 295–296
- Circular *see* recursive
- Clearing the screen, TURBO Pascal 389
- close 509, 609
  - short definition* 534–535
- Closed variable *see* owned variable
- clreol A.64
- clrscr 389
- COBOL 10
- Code 4, 11
- .COM files *see* .EXE files
- Comment 68–69, 73–74, 96, 171
  - short definition* 98
  - see also* assertion; postcondition; precondition
- Comparison rule for strings 294
- Compiler 11–12
  - version 4.0 A.46
  - version 5.0 A.47
- Compiler directive 299, 343, A.72–74
  - B A.73
  - R 299–343
- Compiling to disk A.58–59
- Component *see* array, record
- Component field *see* record
- Component type *see* array, component type
- Compound statement 87, 201
  - short definition* 99
- concat, 44
  - short definition* 321
  - example 44
- const 67–68
- Constant 26, 29
  - declaration 67–68
    - short definition* 97
  - named 67–68
- Control character A.36–37
- copy
  - in DOS A.28
  - in TURBO Pascal 314
    - short definition* 322
- Copy 525
- CPU 6
- CRT 3
- crt 389, A.62–71
- Ctrl *see* control character
- Cursor A.45
- .DAT 609
- Data 9–10
- Data abstraction 388–389, 484–485
  - using units 705–710
- Data flow 56, 123
- Data flow diagram 55–56
- Data representation 205
- Data structure 387–388
  - choosing 438–439
  - hierarchical 439
- Data type 345–348, 387–388, *see also* type
- date A.24–25
- Debugger A.77–85
  - Breakpoint A.83–84
  - Watch window A.79–81
- Debugging 74–77, 250–254, 485–491
  - see also* debugger
  - see also* testing;
  - see also specific topics, e.g., loop, debugging*
- Debugging switch 193
- Decimal Computer 581–585
- Declarations
  - forward 572–574
  - local 151–152
  - order of 108
  - in TURBO 416
  - see also specific topics, e.g., constant, declaration*
- Defensive programming 96
- delay A.62–63
- delete 314
  - short definition* 322–323
- delline A.64
- DeMorgan Law 213
- Design techniques 14–17, 481–485
  - by concrete example 399–404
  - guidelines for 482
  - known algorithms 394, 483
  - see also* analogy (solution by); brute force; data abstraction; data flow; data structure (*subentries* choosing; hierarchical); defensive programming; guilty until proven innocent; iterative enhancement; procedural abstraction; procedure (*subentry* generalizing); problem solving phase; pseudocode; nega-

- tive thinking; recursion; special cases;
- sentinel value; solution space; stepwise refinement; top-down design; unifying cases;
- unrolling a loop
- see also specific tasks, e.g., input, search, etc.*
- Directories A.31–33
- Directory command A.50
- Directory name 507
- Disk 5, 505
  - floppy 5, 6
  - formatting in DOS A.30–31
  - hard 5, 7
- diskcomp A.30
- diskcopy A.29–30
- Diskette *see* disk, floppy
- Display *see* screen
- dispose 670–671
  - short definition* 687
- div 51–53
  - short definition* 61
- Division 29, 51–53
- DOS A.22–35, A.52
- Double precision 588
- downto 247
- Dynamic data structure 636
- Dynamic variable 637–640
  - implementation 671–675
  - see also* node
- Echoing input 38
- Editing out excess blanks 526–529
- Editor 14
  - insert and overwrite modes A.45
  - starting
    - version 4.0 A.41
    - version 5.0 A.44
  - TURBO A.36–45, A.55–57
- Efficiency 492–496
  - versus clarity 495–496
- Element *see* array; set
- else *see* if-then-else
- in case statement 199
- Empty statement 201–202
- End of file
  - unexpected 612
  - see also* eof
- End of line *see* eoln
- E notation 31–32
- Enumerated type 310–312
- eof 229, 518–520
  - short definition* 534, 631
  - pitfalls 522–523
- eoln 227–228, 516–518, 520–523
  - short definition* 260–261, 534
  - pitfalls 522–523
- Equality operator, pitfalls 84–85
- Enter key A.24–25, A.37
- erase
  - in DOS A.28
  - in TURBO Pascal 525
  - short definition* 535
- Error messages 74
  - input/output A.75–76
- .EXE files A.58–59
- Executing *see* running
- exit A.3
- Exponent 31, 34–35, 271–272
- Exponent part 582
- false 188, 193
- Fibonacci number 264
- Field *see* record
- Field identifier *see* record
- Field width *see* formatted output
- File 14
  - binary *see* File, nontext
  - closing 509
  - component type 607
  - data *see* File, nontext
  - DOS A.27–29
  - inputting name 509
  - loading A.49
  - mixing reading and writing 513–514, 526
  - names 505–506
    - in TURBO A.50
    - TURBO Pascal 511–512
  - nontext 606–608
  - opening
    - standard Pascal 512–513, 609
    - TURBO Pascal 507–509, 608–609
  - parameter 523–525, 622
  - path names 512
  - random access 616–619
  - of records 614–616
  - saving A.49–50
  - standard Pascal 512–513
  - temporary 525
  - text 229, 505–506
  - type declaration 607–608
    - short definition* 629
  - variable 506, 608
    - short definition* 532, 629–630
  - what kind to use 616
- File manager 14
- filepos, *short definition* 632
- filesize 617
  - short definition* 631
- Floating point notation 31 *see also* real
- for 244–247
- format A.30
- Formatted output 69–72
  - short definition* 98
  - for real values 70
- For statement 244–247
  - short definition* 261
  - increment size 247
  - type of control variable 299
- FORTRAN 10
- Forward declaration 572–574
- Fractional part 582
- Frame 551
- Friendly programs 239–240

- Fully exercising code 488–490
- Function
  - argument 92–93, 269
  - boolean values 280
  - call 269
    - short definition* 324
  - declaration 269–271
    - short definition* 324
  - defining 269–271
  - heading 270
    - short definition* 324
  - name 273–275
    - see also* function, value returned
  - parameter list *see* function, argument
  - side effects 278
  - standard 92–93, 284
  - type of value returned 348, inside back cover
  - value returned 92, 269
    - changing 277
- Function key A.37
- Garbage collection 672–674
  - see also* dispose, mark, release
- Global constant 153
- Global variable 146–147, 152–153
  - short definition* 175
- goto* A.1–3
- gotoxy A.66–68
- Grade warnings 147–149
- Grading Program
  - using arrays 410–414
  - using records 436–438
- Graphics A.66–68
- Guarded command *see if-then-else*, nested
- Guessing game 50–52
- Guilty until proven innocent 282
- halt A.3
- Hardware 3
- Hashing 451–455
- Headed by size 226
- Heading *see specific topic, e.g.,* program, heading
- High level language 10–11
- Highlighted command A.48
- History table 380–383
- Hotkey A.53–54
  - version 4.0 A.40–41
  - version 5.0 A.43–44
- Horner's rule 501
- Host type 297
- Identifier 38–40
  - short definition* 59
  - local 151–152, 274–275
    - short definition* 175
  - long 39
  - standard 40
  - in TURBO Pascal 42
- if-then* 85–87
  - short definition* 99
- if-then-else* 81–87
  - short definition* 98–99
  - nested 182, 202–203
- Imin 365
  - record version 452
- Implementation phase 14–17
- in* 461
  - short definition* 472–473
  - used with *not* 463
- Including files A.73
- Income tax, state 203–204
- Indenting 46, 96
- Index *see entry under* array
- Induction 556
- Input 3, 9–10, 36–38
- Input/output, redirection A.60–61
- insert 314
  - short definition* 323–324
- inline* A.64
- integer 28, 31, 293
  - short definition* 60
  - largest value *see* maxint
  - machine representation 581, 586–587
  - related types in TURBO Pascal A.5
- Invariant *see entry under* assertion
- Invocation *see* procedure call, function call
- I/O error handling A.75–76
- I/O redirection A.60–61
- ioresult A.75–76
- Iterative enhancement 88
  - example of 88–92
- Jump *see goto*
- Keyboard 3
- keypressed A.64
- Label
  - case statement 197
  - used with *goto* A.1
- Label list 197
- Largest and smallest number, finding the 235–237
- Leap year 172–173
- length 314
  - short definition* 321
- Length 395–397
- Life (game of) 421
- Linear congruence method 286–287
- Lines 41
- Linked list 645–649
  - doubly linked 676–678
  - empty 650, 662
  - procedures for manipulating 651–659
- Listing 77
- Local variable 145–147
  - short definition* 175
- Logical error 75–77, 487
- longint A.5
- Loop 218–261
  - body 219–220
  - designing 235–239



- common errors 250
- debugging 250–254
- documenting 242
- infinite 223–224
- initializing 235
- iteration 219
- nested 250
- repeat N times 248–249
- terminating 225–226
- tracing 252–254
- what kind to use 249–250
- see also* while statement; repeat statement; for statement
- Lovelace *see* Augusta (Ada)
- Low level language 10–11
- Machine language 11
- mark 675–676
  - short definition* 687
- Match 400
- Matrix 422
- maxint 72–73, 581
  - short definition* 98
- Memory 3–5
  - auxiliary *see* memory, secondary
  - main 5
  - secondary 5, 505
- Menu
  - File A.49–52
    - version 4.0 A.39–40
    - version 5.0 A.42–43
  - main
    - version 4.0 A.38–39
    - version 5.0 A.41–42
  - pull-down
    - version 4.0 A.39–40
    - version 5.0 A.42
  - TURBO 4.0 menu environment A.38–41
  - TURBO 5.0 menu environment A.41–44
- Merging two files 622–627
- Micro computer 13
- mkdir A.32
- mod 51–53
  - short definition* 61
- Modula 10
- Monitor *see* screen
- Multiplication 26
- Name *see* identifier
  - choosing 46
  - for constant 67–68
- Negative thinking 528
- Nested data structures *see* data structure, hierarchical
- Nested statements 181–182
- Network 13
- new 639
  - short definition* 686–687
- nil 644–645
  - short definition* 687
- Nim 327
- Node 643–644
  - lost 650–651
- nosound A.64
- not 183
  - short definition* 212
- Object program or code 11
- Off-by-one errors 250
- Off-line data 229, 362, 496 *see also* files
- One-way parameter *see* value parameter
- Operating system 13, *see also* DOS
- Operator
  - arithmetic 33–34
  - relational 83–85
  - see also* precedence rules
- or 183
  - short definition* 211–212
- ord 295–296
- Ordinal type 293–294
- Output 3, 35–36
  - designing 191–193
- Overflow 75, 583
  - stack 551–554
- Own variable *see* owned variable
- Owned variable 700–702
- packed* 404–405 *see also* arrays, packed
- Packed array of characters *see* array
- Parameter 107–112
  - actual 109, 122
    - short definition* 138
    - expressions as 122
  - comparison of variable and value parameters 124
  - formal 109, 122
    - short definition* 137
  - names 134–135
  - substitution 109–113
  - summary of kinds 124
  - type conflict 301–302
  - value 121–123
    - short definition* 138
    - implementation 155
    - as local variable 155–157
  - variable 107–109
    - short definition* 138
    - implementation 112–114
    - what kind to use 123–124
- Parameter list 109–112, 124–125
  - incorrectly ordered 132–133
- Parentheses 33–34
  - in boolean expressions 184, 189
- Pascal, Blaise 47
- Pattern Matching 390–401
- Payroll calculation 88–92
- Perfect number 264
- Personal computer 13
- Plus sign, with strings 43
- Pointer 637–643
  - with := 641–642
  - function 661–662
  - implementation 671–675

- types 638
    - short definition* 686
  - using recursion on 661–662
  - variable 638
  - Pop 667–668
  - Portability 39, 296, 492
  - pos 313
    - short definition* 321–322
  - Postcondition 171, 243
  - POW 546
  - Power 271–272
    - recursive version 544
  - Powers *see* exponent
  - Precedence rules 34, A.6
  - Precondition 171
  - pred 295
  - Pretty print 577
  - Prime number 280–283
  - Printer 3
  - Private variable *see* owned variable
  - Problem solving *see* design techniques
  - Problem solving phase 14–17
  - Procedural abstraction 118, 145, 250, 388, 484–485
    - using units 698–700
  - Procedure
    - body 107
    - call 107–109
      - short definition* 137
    - within a procedure 114–118
  - declaration 107–109
    - short definition* 136–137, 174
  - generalizing 133–134
  - heading 107, 125
    - short definition* 137
  - local 151–152
  - simple 107
  - testing 165–170, 233–234
- Production graph 357–359
- Program 8–13
  - body 40–41
  - heading 40
    - in TURBO Pascal 41
- Programming language 8
- Prompt line 38
- Pseudocode 51
- Push 667–668
- Quadratic equation 93–96
- Queue 420, 706–710
- Quoted string 61 *see also* quotes
- Quotes 29, 35
  - inside of quotes 35
- Random 287
- random 292
- Random number generator 284–291
  - real valued 287
  - scaling values 287–290
  - TURBO Pascal 290–291
- RandomReal 287
- R compiler directive 299, 343
- read 36–38
  - short definition* 61–62, 533, 630
  - with nontext files 611
  - with text files 507, 517
- readkey A.65
- readln 36–38
  - short definition* 61, 533
  - with text files 507, 517
- real 28–29, 31–32
  - short definition* 60
  - equality of 224, 582
  - largest value 72–73, 582–583
  - machine representation 581–582, 587–588
- Record 427–434
  - short definition* 470
  - : = used with 429
  - comparison to arrays 431–432
  - syntax 432–433
  - variant 455–461
    - short definition* 471–472
- Recursion 181, 542, 545–547
  - functions 543–547
  - infinite 547–549
  - compared to iteration 565–567
  - mutual 574
  - stopping case 547, 556
- Redirection A.60–61
- Relational operators 83–85
- release 675–676
  - short definition* 688
- rename A.28–29
  - short definition* 535
- repeat 230
- Repeat statement 230–231
  - short definition* 260
  - comparison to while statement 231
- Reserved word 40, inside back cover
- reset 509–511, 609
  - short definition* 532, 630
- rewrite 508, 609
  - short definition* 532, 630
- rmdir A.32
- Robust programs 239–240
- Roman numerals 216
- Root of a function 594–599
- round 93
- Rounding to zero 590
- Round-off error 590–593
- Running a program 9–10
  - version 4.0 A.46–47
  - version 5.0 A.47–48
- Run-time error 75–76, 486
- Safran, Olivia 289
- Sales report 441–442
- Saving a file A.49–50
- Scientific notation 31

- Scope 157–162
  - short definition* 175
- Scope rule 158
- Screen 3
- Search
  - binary search 567–572
    - on trees 681–682
  - serial 390, 391–392, 493–495
  - see also* hashing
- Seed 286
- seek 616–617
  - short definition* 631
- seekeof, *short definition* 536
- seeko1n, *short definition* 535
- Semicolon 23–24, 201–202
  - in *if-then-else* 87
  - see also* empty statement
- Sentinel value 226
- Serial search *see* Search, serial
- Series (summing) 247–248, 593–594
- Set 459–469
  - constants 468–469
  - elements of 464
  - empty 467
  - integers in 464
  - limitations on 464
  - operators 466
  - in TURBO Pascal 464
  - type definitions 465
  - short definition* 473
- shortint A.5
- Side effect 152–153, 278
- Simple type 295
- Software 3
- Solution space 58
- Sorting 362–364
  - a file 663–667
  - insertion 663–667
  - interchange 363
  - records 449–450
  - selection 363
- sound A.63
- Source program or code 11
- Spacing 41
- Special cases 529
- sqr 93
- sqr t 93
- Stack 550–551
  - implemented as linked list 667–670
- Statement 23–24
- Static variable *see* owned variable
- Stepwise refinement 50
- Storage *see* memory
- str 232
- String 67, 375
  - comparisons in TURBO Pascal 294–295
  - constants 67
  - implemented as records 442–449
  - manipulation, in TURBO Pascal 313–317
  - names 507, 609
  - parameters 300–301
  - types 375
  - variables 42–46
  - see also* arrays (*subentry* packed array of characters)
- string 42
- string processing, in TURBO Pascal 313–320
  - short definition* 321–324
- StringReadln 376
  - record version 447
- StringWriteln 376
  - record version 448
- Stub program 168
- Subpattern 402
- Subrange type 296–299
  - short definition* 325
  - for error checking 298–299, 307
- Substring, replacing 315–317
- Subtask 50–51
- succ 295
- Supermarket pricing 126–128
- Syntax 74, 79–80
- Syntax diagram 79–81, 182, A.7–21
- Syntax error 74–76, 486
- Systems programs 12
- Tag field *see* record, variant
- Tape 5, 7
- Temporary files 525, *see also* erase
- Terminal 3
- Testing 15–17, 74–77, 485–491
  - all paths 489–490
  - using assertions 78–79
  - bottom-up 165–169
  - procedures 165–170, 233–234
  - top-down 165, 169
  - see also* debugging; fully exercising code; tracing
  - see also specific topics, e.g., loop*
- text 505–506
- Text file *see* file
  - editing 525
    - blanks 526–529
    - as a programming aid 529
- Threshold *see* variant expression
  - for recursive function 555–556
- time A.24–25
- Time-sharing 13
- Top-down 165, 169
- Top-down design 50
- Towers of Hanoi 560–565
- Tracing 77–79, 252–254, 491
- Tree 678–682
  - balanced 685
  - binary search trees 681–685
  - empty 680–681
  - inorder traversal 678–680
  - postorder traversal 681
  - preorder traversal 681
  - root 678

- true 188, 193
- trunc 93
- truncate, *short definition* 632
- Truth tables 186
- TXT 507
- Type 28–33, 345–346
  - anonymous 302–303
  - declaration 297–299, 338–339
  - structured 346–348
  - see also specific types, e.g., integer*
- type
  - in DOS A.28
  - in Pascal 297–298, 300, 338–339
- Type compatibility 32–33, 297–298, 301–302
- Typed constants 412–415
- Underflow 583–584, 590
- Underscore symbol 42
- Unifying cases 205
- Uninitialized variables 27, 222–223
- Unit 691–714, *see also crt*
  - Bebugger A.85
  - compiling 695
  - data abstraction 705–710
  - implementation section 693–695
  - initialization section 700–702
  - interface section 693–695
  - owned variable 700–702
  - procedural abstraction 698–700
  - short definition* 711
  - unit directories 703–705
  - uses clause 695–698
- Unrolling a loop 237–239
- upcase 296
  - short definition* 322
- Uppercase letters 39, 240
- Uses clause 695–698
- val 232
- Value 23, 25–26
- Value range error 307
- var 28–30, 109–110, 124, 145–147, 157–158
- Variable 23, 25–28
  - boolean 188–189
  - controlling a for statement 245
  - declaration 28, 30
    - short definition* 59–60
  - indexed *see* array, indexed variable
  - initializing 27, 235
- Variant expression 242–244
  - for recursive function 555–556
- Verification 491
  - for recursive function 555–556
- Watch window A.79–81
  - clearing A.83
  - expressions in A.80
- wherex and wherey A.64
- while 220
- While statement 220–222
  - short definition* 259–260
  - comparison to repeat statement 231
- Wildcard A.29, A.50–51
- Window
  - in 4.0 environment A.39
  - in 5.0 environment A.42
  - in nontext file 609–610
  - in TURBO Pascal program A.69–71
  - zoom and unzoom A.44–45
- window A.69–71
- Wirth, Niklaus 13
- with 439–440
  - short definition* 472
- Word 3, 581, 586
- word A.5
- Work file A.49–50
- write 35–36
  - short definition* 61, 533, 630–631
  - with nontext files 610
  - with text files 506
- writeln 35–36
  - short definition* 61, 533
  - with text files 506, 516–517
- Zoom A.44–45





























---

## Reserved Words

*\*absolute*  
*and*  
*array*  
*begin*  
*case*  
*const*  
*div*  
*do*  
*downto*  
*else*  
*end*  
*\*external*  
*file*  
*for*  
*forward*  
*function*  
*goto*  
*if*  
*\*implementation*  
*in*  
*\*inline*  
*\*interface*  
*\*interrupt*  
*label*  
*mod*  
*nil*  
*not*  
*of*  
*or*  
*packed*  
*procedure*  
*program*  
*record*  
*repeat*  
*set*  
*\*shl*  
*\*shr*  
*\*string*  
*then*  
*to*  
*type*  
*\*unit*  
*until*  
*\*uses*  
*var*  
*while*  
*with*  
*\*xor*

## Ordinal Types

integer, char, boolean, longint\*,  
shortint\*, word\*, byte\*  
any enumerated type, any subrange type.

## Types that May Be Used for the Value Returned by a Function

All the ordinal types.  
The type *real*.  
Any pointer type.  
The *string* types.\*  
No other types may be used.

## Ranges for TURBO Pascal Numbers

maxint = 32767  
smallest negative value of type integer = -32768  
largest value of type real = 1.0E38  
smallest positive value of type real = 1.0E-38

*\*In TURBO Pascal but not in standard Pascal.*

---



## Predefined Functions

| NAME   | DESCRIPTION       | TYPE OF ARGUMENT | TYPE OF RESULT  | EXAMPLE                                   | VALUE OF EXAMPLE               |
|--------|-------------------|------------------|-----------------|-------------------------------------------|--------------------------------|
| abs    | absolute value    | integer<br>real  | integer<br>real | abs (-2)<br>abs (-2.4)                    | +2<br>+2.4                     |
| round  | rounding          | real             | integer         | round (2.6)                               | 3                              |
| trunc  | truncation        | real             | integer         | trunc (2.6)                               | 2                              |
| sqr    | squaring          | integer<br>real  | integer<br>real | sqr (2)<br>sqr (1.100)                    | 4<br>1.2100                    |
| sqrt   | square root       | real or integer  | real            | sqrt (4)                                  | 2.00                           |
| arctan | arctangent        | real or integer  | real            | arctan ( 1.0 )                            | 0.785<br>(radians)             |
| cos    | cosine            | real or integer  | real            | cos ( 0.78 )<br>(argument in radians)     | 0.71091                        |
| sin    | sine              | real or integer  | real            | sin ( 1.57 )<br>(argument in radians)     | 1.00                           |
| exp    | exponential       | real or integer  | real            | exp ( 2 )                                 | $e^2$<br>(not Pascal notation) |
| ln     | natural logarithm | real or integer  | real            | ln (2.71828)<br>( $e$ is approx. 2.71828) | 1.000                          |
| odd    | odd/even function | integer          | boolean         | odd (5)<br>odd (4)                        | true<br>false                  |



**The Benjamin/Cummings Publishing Company, Inc.**

Redwood City, California • Fort Collins, Colorado

Menlo Park, California • Reading, Massachusetts • New York

Don Mills, Ontario • Wokingham, U.K. • Amsterdam • Bonn

Sydney • Singapore • Tokyo • Madrid • San Juan

### **TURBO Pascal 4.0/5.0:**

#### **An Introduction to the Art and Science of Programming**

If you've recently updated your system to either version 4.0 or 5.0 of TURBO Pascal, this is the book for you. All the programs in the book run on both versions of TURBO with differences in keystrokes and commands clearly spelled out. Plus units, windows, the interactive debugger, and other important new features are covered in detail. You will learn how to write separately compilable units to implement data abstraction. And you'll also develop excellent testing and debugging techniques using the new interactive debugger.

**TURBO Pascal 4.0/5.0** is the latest in the series of highly successful Pascal books by Walter J. Savitch. As with all of Professor Savitch's books, you'll find a strong emphasis on problem solving and an excellent, accessible introduction to modern programming techniques. In addition, the book includes a unique chapter on numeric programming and optional in-depth coverage of advanced Pascal topics including recursion, pointers, and dynamic data structures.

#### **Key Features:**

- ☐ Complete appendix to the "windowing environment"
- ☐ Detailed explanations of data abstraction and implementation techniques using units
- ☐ Thorough coverage of the new interactive debugger with practical guidelines for debugging and testing a program
- ☐ Detailed discussion of compiler directives and string handling
- ☐ 90 substantial program examples in an innovative case study format
- ☐ Extensive appendices to DOS commands and TURBO features

#### ***Books of Related Interest:***

**TURBO Pascal: An Introduction to the Art and Science of Programming, Second Edition** (38396) By Walter J. Savitch

**Pascal: An Introduction to the Art and Science of Programming, Second Edition** (38388) by Walter J. Savitch

**Computer Science: An Overview, Second Edition** (30903)  
by J. Glenn Brookshear

Savitch



# TURBO Pascal 4.0/5.0

*An Introduction to the Art  
and Science of Programming*



30410